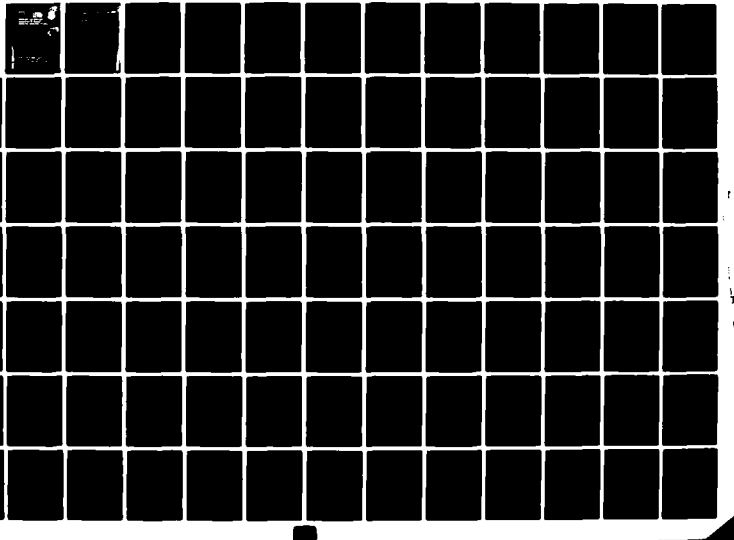
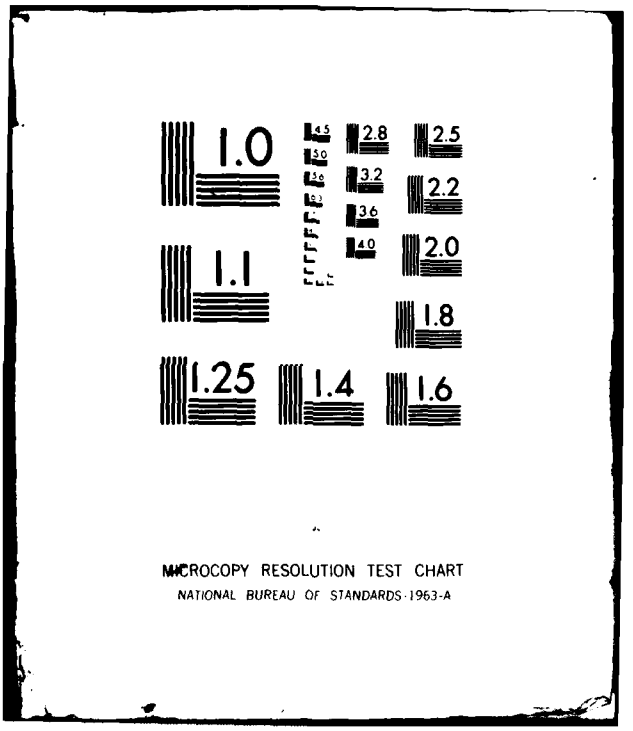
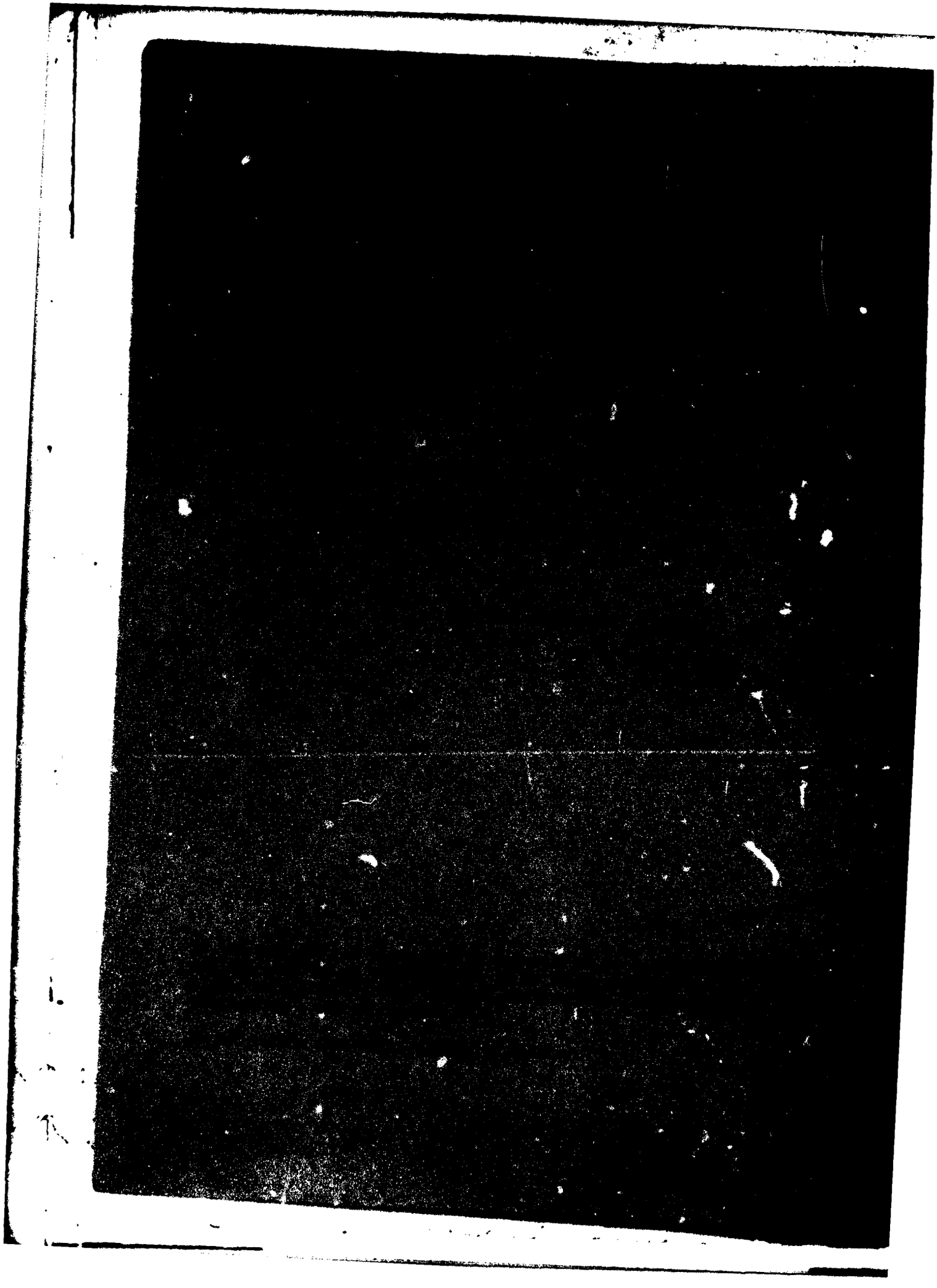
AD-A082 363 WAYNE STATE UNIV DETROIT MICH F/G 9/2
MODELLING AND RESOURCE ALLOCATION OF LINEARLY RESTRICTED OPERAT--ETC(U)
DEC 79 T FENG, C P HSIEH F30602-76-C-0282
UNCLASSIFIED RADC-TR-79-311 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: RADC-TR-79-311	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): MODELLING AND RESOURCE ALLOCATION OF LINEARLY RESTRICTED OPERATING SYSTEMS	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. May 1976 — June 1979
	6. PERFORMING ORG. REPORT NUMBER: N/A
7. AUTHOR(s): Tse-yun Feng, C. P. Hsieh	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0282
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Wayne State University, Detroit MI 48202	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55971405, 12 156
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISCA), Griffiss AFB NY 13441	12. REPORT DATE: December 1979; 13. NUMBER OF PAGES: 239
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED; 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: James L. Previte (ISCA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Operating Systems
Queues
Resource Allocation
Linear Programming
Computer Systems

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Operating System is an integral part of a total computing utility. Its structure is complicated and the viewpoint on the subject is diversified. If the computing system is viewed upon as a collection of different resource types to serve different users with different demands, then the operating system assumes the managerial role. To best utilize the available resources to achieve a desirable level of production, i.e., computation, an optimal planning (programming) is needed. Optimality can be judged only after a performance

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

JOB

index can be established. Operating System viewed as a conglomerate of interdependent activities is similar to that of an economic microcosm. The question of system resource allocation is then formulated as a Linear Programming Problem with constraints on resources and optimization is over a linear objective function. This general objective function can be stated only after the view on the performance question has been broadened. Program loading (memory allocation) is static while program execution (scheduling) is dynamic in nature, in a multiprogram, batch environment. The dynamic problem is studied through the viewpoint of memory utilization, with a warehousing model. Finally, user program characteristics under page-on-demand environment is studied; a possible analytic representation of this property and its implication on memory allocation is suggested.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

EVALUATION

An operating system is difficult to define and its design specification cannot be quantified, if it can be specified at all. This study has sought to develop some common rules in system design rather than an implementation on gained experiences or application requirements. As such the discussion is general wherever possible so that the considerations may be applicable to a wide range of situations. The resource allocation question is particularly addressed, especially memory allocation in a multiprogrammed environment.

This work is of considerable interest to TPO R3D. Knowledge gained from this work will be factored into future distributed data processing investigations.

JAMES L. PREVITE
Project Engineer

CHAPTER I

Introduction

The degree of sophistication in today's computer systems is a far cry from those programmable numerical integrating devices of yesteryear. The advances in hardware capabilities and complexity necessitate some form of automated procedures in operating them. These automated procedures are implemented in a collection of programs. We customarily think of programs as software. In fact, a hard-wired schematic is also a form of programming but with considerably less flexibility.

Our ideas on how to automate the system operations change often as we learn from experience; the process is necessarily an evolutionary one. In this light, flexibility is more than just a plus; it is sine qua non. And today, whenever we speak of operating systems, the mind immediately conjures up the image of programs -- software programs at that. This perception is not far off, even though cases can be made where the operating system actions are furnished by hardware. However, one should not confuse the notion of an operating system with that of system programming; the latter is but implementation of the concepts of the former. These concepts and ideas are the culmination of much trial and error over the years. The specific implementations are mostly drawn from personal experiences. In spite of the constant association (everyone who uses computer today would come into contact with an operating system of one form or another, unless it is a more primitive one, like a microprocessor) and constant efforts in this direction in recent years, we still do not understand very much about the operating system. Oftentimes, we are still "amazed" by the wholly "unexpected" change in system character by some seemingly innocent changes in one of the system modules. We do not yet have any viable "theory" pertinent to the operating system, and even its "principles" are sometimes haphazardly stated. Nowhere more glaringly apparent is the fact that the terminologies are oriented toward specific system implementation than uniformity. Standardization would certainly be the first step in the appropriate direction.

Unlike some other branches of the system sciences, an operating system is one that is difficult to define, and its design specification cannot be quantified, if it can be specified at all. But at least we should search for some common rules in system design rather than an implementation based on gained experiences or application requirements. Using this as a guiding

principle, our discussions that follow will adhere to generality whenever possible, so that the results may be applicable in a wide range of situations. Specifically, we shall study the resource allocation question in general and the memory allocation question in particular in a multi-programmed environment. In carrying on the discussions, we have used the terms program, job, task, process interchangeably, whenever the meaning is clear in the context.

In order to see where we might be going, we should know where we have been. Chapter 2 gives a brief historical review on the subject of operating systems, dating back to the days when it was only known as an in-core supervisor. Although some of the important technical developments were precipitated by specific events or specific systems, we have avoided introducing specific models of particular manufacturers into the picture. The continuity in thought has not been compromised by this approach, we believe.

Chapter 3 continues to examine the modern systems. Out of many areas of system abstractions, we concentrate on the concurrent processes, queueing models, and working set model. Even though these three areas are outwardly quite different, they are drawn together by a common concern, i.e., they are all related to system resources; either they deal with them or are defined by them, explicitly or implicitly. Finally, the question of performance is re-examined and the view broadened.

In Chapter 4, we introduce the concept of activity aggregates. Some basic properties of system activities are postulated. If we view the entire operating system as being one which is a conglomerate of individual activities, comprised of both the system's own and those of the users', and if the concept of computing utility can be interpreted as the availability of a primary "commodity" for individual "production needs", then the function of an operating system becomes a managerial one. The system resource allocation policy becomes that of determining an optimum plan for distributing the limited resources under linear constraints. A specific activity, the memory allocation activity, is formally stated in the context of Linear Programming. The Simplex Method for solving Linear Program Problems is then briefly described.

Chapter 5 probes deeper into the question of memory allocation in the context of program loading (selection from the system job queue). Furthermore, the result of this selection can be controlled by the appropriate choice of an objective function. A generalized value concept is then introduced in

conjunction with the formulation of linear objective functions. This generalized approach embodies the priority scheduling. We also differentiate the term scheduling from allocation. The point is brought into sharper focus by discussing a dynamic formulation of the loading and executing activities. While loading reflects the resource allocation policy, executing activity manifests the policy of scheduling.

In Chapter 6, we again examine the question of memory allocation in a paged environment. The addressing characteristics of programs (user profiles) are discussed, using the concept of a Lifetime Function propounded by Belady and Kuehner as a vehicle. And the possibility of using this Lifetime Function characteristic as a criterion for initial memory allocation is explored; an analytic form of this function is suggested, and a curve-fitting procedure is also proposed for obtaining the unknown parameters from measured data points.

In Chapter 7, we conclude that by carefully defining the activities based on the concepts set forth in Chapter 4, and with a broadened view on performance, general principles for optimal policies can be studied through the aid of mathematical programming.

CHAPTER II

Operating System in Perspective

2.1 Introduction

In this chapter we will first trace back some of the history to see where it came from and how it came about. In so doing, hopefully we can see where it should be going. Section 2 gives a historic view on the subject. No particular machine will be mentioned, but, the emphasis is rather on the motivation and the hardware/software interaction which influenced the conceptual and technical developments of what came to be regarded as operating system. Section 3 gives an overview on the scope and capabilities of today's system. Finally, Section 4 gives a brief discussion on our view; it is of a managerial nature.

2.2 Historic View - A Précis

Ever since the first large scale electronic computer, ENLAC, was completed in 1946, computers have progressed from programmable calculating devices to the myriad of computing facilities within a span of three decades. Certainly the concept of a computer has also evolved from high speed (relatively speaking) numerical computational tool to that of a general purpose computing utility. The users today have come to view the entire hardware/software complex as an "environment" within which the computational tasks were carried out. Of course it did not begin this way; the early machines had no software support at all. In studying the chronology of electronic computers, the term "generation" has been widely in use since the mid-60's. Mainly, it is used to characterize the stages of hardware development. They are roughly as follows [1]:

First: 1946 - 1959

Second: 1959 - 1965

Third: 1965 - onward

The above dates are approximate ones but there is little disagreement on what belongs to which generation; for they are so divided according to the technology of implementations. Namely, the first generation was a vacuum tube era. The second generation was characterized by transistorized logic. Integrated circuit technology ushered in the third generation. However, since the announcing of IBM System 370, there has been no consensus as to whether the fourth generation has arrived. For there is no clean cut boundary, either in

terms of technology or concepts, which might separate them from the third generation, as did the previous ones. In fact, it is a system of evolutionary refinements. Some have thus labelled the years from 1968 until present time as the late Third Generation.

There were also efforts in dividing the software developments and systems into "generations". It is less successful because of the circumstances surrounding the software developments. The term "software" here does not refer to user, or the so-called application programs. It is the facility with which programmers can efficiently develop, debug and execute their own programs. Initially, none existed. With the hardware advances and therefore the desire for their better utilization and better service to users, the concept of certain controlling routines evolved, which again led to certain added hardware feature and improvements, etc. Therefore, we may say that software maturity follows the hardware development and in turn acts partially as an impetus for the coming into being of a certain special purpose hardware. This sort of "development cycle" is at least true for the earlier generations of computers.

According to Rosin [2], there are periods of development that can be identified with their capabilities. Chronologically, they are:

Pre - 1956: Job-By-Job Processing
1956 - 1958: Early Batch System
1959 - 1961: Executive Systems
1962 - 1964: Upgrading and Higher Volume
1965 - 1968: Refinement

In each period, there were innovations (technical as well as conceptual) except, of course, the Pre-1956 days. In tracing through these tidbits, we begin to get a picture of the hardware/software interactions behind the development forces.

From the beginning, the majority of programs were written in machine code, in either decimal or octal, in absolute rather than relocatable mode, and with no library routines; or in a primitive assembly language which allowed the user to assemble symbolic library routines, provided in a card deck, along with his program. All I/O devices, namely, card reader and line printer (maybe just the console typewriter) were on-line. The user had to operate the computer and the peripherals himself. Other than perhaps a self-loading one-card loader, which occupied very little memory space, practically

all the information in memory was put there by the user himself. By using the console switches and lights, the user was able to trace through and to understand precisely the status of his program executions. In a way, the user had complete contact with the machine.

As the hardware speed and cost increased, it was obvious that the previous arrangements were not desirable from the economical point of view. It had gotten to the point where many of the manual procedures took much longer time than the actual program assembling and execution. It appeared that the multi-step procedures such as loading an assembler, assembling a program, and then reloading the object code, could be automated by the computer itself under the control of a specially constructed program. The advent of FORTRAN in 1957 provided additional motivation; now the user had a choice between calling either the assembler or the FORTRAN compiler. The slow on-line card reader and line printer were circumvented by using magnetic tapes. Jobs were grouped together into a collection called a batch. The batch was read onto a tape by an off-line card-to-tape device. A small controlling program, called a monitor, performed the tasks of sequencing from job to job, calling in various system components as needed: assembler, compiler, loader (object program), along with subroutines from a library file on tape and perhaps also a post-mortem dump routine. Printed output was processed from the output tape by an off-line tape-to-printing device. I/O tapes were hand carried. Whatever special manual operations or instructions such as handling user tapes were performed by an operator. The user, in general, was by then totally removed from direct physical contacts with the computer. Fig. 2.1 depicts the structure of such a facility.

The assignment of special tape units for I/O purposes meant that there must be I/O standardization. Every user had to program all Read/Write operations using these units and was forbidden to manipulate those tape units assigned for system use. Furthermore, the I/O tapes contained more than one user. Care had to be exercised to prevent reading from the input tape beyond one boundary of one's own data and to prevent writing over someone else's result on the output tape. To reduce such unhappy occasions, the system library also contained a set of standard I/O routines for returning control to the system at the end of file tape mark, and for requesting a dump before signing off. FORTRAN programs had to use these routines since the language itself had no such facilities. To make it easier for assembly programs to use

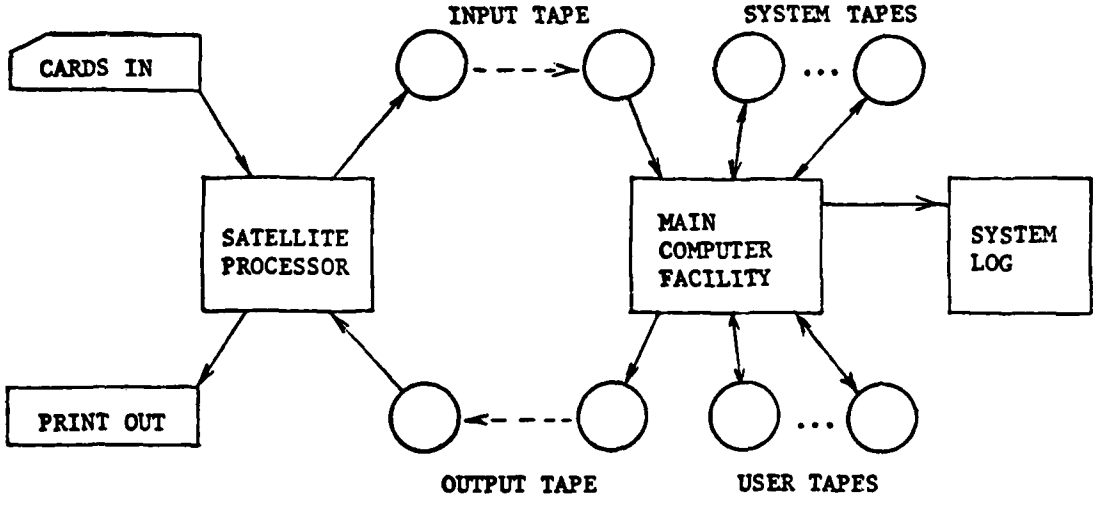


Fig. 2.1 Batch System with off-line I/O

these routines and also those system supplied subroutines primarily intended for FORTRAN programs, such as square root, etc., standard subroutine calling sequence and linkage were established.

Because of the increasing flexibility in linking subroutines and in order to make effective use of high speed storage space, all object programs were in relocatable mode rather than absolute. The loader then would have to link the collection of relocatable object programs into a well defined absolute program. Also, the limited amount of memory space led to the development of job step chaining or overlay. Finally, since the user was totally absent from the machine, and most of the activities within the computer were automatic, the on-line printer (or console typewriter) was assigned the function of recording the system log so that it at least provided a way to monitor the behavior of the system, and for communicating with the operator. The availability of a system clock made it possible for the accounting function to be partially automated. Many of the concepts and techniques developed in this period of early batching systems really laid down the groundwork for the succeeding executive systems and the refinements and extensions later on.

During the years from 1959 to 1961, certain hardware developments were most significant. It was the data channel, and later, the ability of the channel to interrupt main frame activity. The data channel was essentially what we would call a peripheral processor today. Their usage allowed overlapped I/O and computational processing. A buffer facility was used, and a wait loop was needed to test the I/O activities. The advent of I/O completion interrupts and exception operations (end-of-file, end-of-tape, etc.) allowed fully overlapped I/O buffering without the wait loop. The interrupt enabling, masking, and disabling created a very complex situation. This necessitated a set of standard I/O request and interrupt handling routines. The mandatory usage of these packages led to the establishment of permanently resident I/O supervisor and "low core" or interrupt processor. This created a problem in the area of protection. There was no sure way of preventing the resident supervisor being overwritten except with hardware protection facilities.

The generality provided by the centralization of I/O processing and system supervision dictated a high degree of modularity in the overall system and its components. It became clear that all system components should be addressed through standard interface routines. All the components would communicate with each other using a fixed communication region. Finally,

the diversity of available facilities and functions demanded the development of consistent and convenient methods for making use of the system. Thus a command language evolved. We see that the major techniques and important concepts had been developed in this period which we would consider as the coming of the Executive System. The years that followed were essentially a case of upgrading and refinement. But several important hardware developments should be pointed out. Hardware protection in the form of a programmable interrupting clock and hardware storage protection allowed in the detection and halting of an infinite loop in a program and permitted the integrity of the resident supervisor to be guaranteed. An important advance was the arrival of the magnetic disc. With its relatively short access time and high data capacity, coupled with the increasingly fast and larger memory space, the impact on the design of supervisory systems was tremendous.

2.3 Modern Operating Systems

Whenever we think of the modern system, the mind immediately conjures up the image of a facility that, with a combination of hardware and software, provides the user a great deal of flexibility in using it. It is quite common to have a system that supports multiprogramming, local and remote batch processing, time-sharing, extensive file manipulating capability and on-line text editing, etc. There are some important developments in system concepts that mark it as state of the art.

The provisions for user's data and program files to be resident in the system on direct access devices, i.e., drum, disc, and on sequential devices such as magnetic tapes, together with general accessing routines and message generators (for counting and dismounting instructions to operator) made it possible for users to build large and complex files for use in their programs without the drudgery of programming those routines.

In supplementing the hardware protection facilities discussed above, the interrupt facilities were expanded to cover the execution of some potentially dangarous instructions such as: illegal op-code; setting the internal timer; setting the protection register, etc. This further led to the concept of master/slave mode of operation. Any interrupt would cause the hardware to be in a master mode, i.e., the supervisor will get control of the processor. Thus, the interrupt is the only facility with which a user can communicate with the system and also, the only way for the supervisor to get control of

the processor back from the user. In dividing up the system into a privileged and an unprivileged class of operations, the supervisor has become the final arbitrator.

In order to support more batch or interactive terminal users, a paging scheme was devised. This came about from the observation that at any given time, only portions of the program address space were needed in memory to sustain the execution. The ability of a program to address a virtual store much larger than a physical store meant that additional hardware for dynamic address translation and additional supervisor routines were necessary.

Operating systems have evolved along with the increasing sophistication of hardware capability as well as the concept of computing utility. If, looking at them from the point of view of their characteristics, the present day systems have the following common properties:

Concurrency
Resource allocation (automatic)
Resource sharing
Multiplexing
Remote access (batch or conversational)
Nondeterminacy
Long-term store

These properties are not completely independent from each other; and no particular system need exhibit all of them. The advent of the data channel made it possible to overlap CPU and I/O processings and therefore multi-programming. If we look at a system with one CPU and several channels, under multiprogram configuration, most likely there will be several processes being "active concurrently" at any given time. Since different uses present different demands on limited resources, their allocation would have to be automatic and centralized. Not all physical resources can be shared but there are non-physical resources in the form of system routines, such as re-entrant codes and data files which are intended to be shared with others. To accomplish this sharing, the system must provide time-multiplexing, e.g., the sharing of CPU in time-sharing system. And the nondeterminacy is a necessary consequence of concurrency, sharing, and multiplexing. That is, the order in which resources are assigned, released, accessed, or shared by concurrent processes is unpredictable and the system routines controlling these activities must make provision for this.

2.4 Summary and Discussion

In previous sections, we have pieced together a view of the evolution of operating systems and the motivations behind these developments. The fact that, in earlier times, they were termed supervisor, or monitor, really conveyed their intended functions. After those early trial days, the operating system has come into its own together with the third generation hardware systems. The operating system has matured in the sense that we pretty much know what the system "should" be capable of doing in a computing facility and the technical know-how to somehow producing it. That is not to say that we know how to make an operating system work very nicely. Can we define an Operating System? Denning [3] defined it through the functional characteristics of computer systems. A computer system has to perform the following seven functions:

1) Creating and removing processes
2) Controlling the progress of processes
3) Acting on exceptional conditions arising during the execution of a process
4) Allocating hardware resources among processes
5) Providing access to software resources
6) Providing protection
7) Interprocess communication.

The system software that assists the hardware in implementing these functions is called the "Operating System". Defined in this way, an operating system is being viewed as the software complement to the hardware to make a complete computer system. However, it does not have to be viewed this way. There are certain system functions which can be thought of as software with hardware assists; the dynamic address translation of virtual system being a case in point. To define something through its functional characteristics often runs the risk of an incomplete definition if some functions are left out. In the case of computing systems, we actually have the situation in reverse; namely, that although some of today's smaller systems do not perform all seven of the above functions, yet there is something in the package which we would unmistakenly identify as an operating system.

Formalism, or a "theory" of operating systems is clearly lacking. The implementation of a system had, in the past, been more or less drawn from the designer's own accumulated experiences, strictly on ad hoc basis. To a large degree, it still does today.

Perhaps we should look at the operating system as being part of a total system that consists of mostly software system routines, together with some hardware complements, performs controlling functions and carries out decision making processes. Hardware components become part of the resources under the aegis of the operating system and together they form the complete computing utility in the eyes of the user.

CHAPTER III

Resource Related System Abstractions

3.1 Introduction

In the previous chapter, we have discussed and recognized that although the system software began its life subservient to the system hardware, the third generation of computer system really established the synergism of these two. Intense effort has been directed at the studying of various aspects of the Operating System. The path through the labyrinth to the central chamber of understanding has not yet been found; we still cannot define the operating system satisfactorily. However, abstraction is our intellectual tool for coping with complexity. Many areas of abstraction have evolved. Some are concerned with its complexity [4,5]. Others consider the design techniques and correctness [6,7]. Still others consider the system from its design objectives [6,7,8,9]. But the most viable ones appear to be in the areas of programming, resource allocation, concurrent processes and protection. We will single out three specific areas for discussion in this chapter. They are: concurrent processes; queueing model; working set model. The reason is that all three are either implicitly or explicitly connected with system resources. A process is considered to be the smallest unit of work that needs system resources. Queueing theory and models deal with scheduling algorithms; scheduling in the sense of controlling the flow of processes and not of actually allocating any resources. Processes will "queue-up" as the computation demands. Working Set Model attempts to view a specific resource allocation problem, i.e. memory management in a paging machine, through user program properties.

In section 3.2, we discuss the various states a process may encounter throughout a computation. Because of the various resource demands, a deadlock situation could happen if we do not place any condition on granting requests. In section 3.3., we begin by examining a single queue model in fairly detailed manner. Some of the current studies in queueing are then briefly reviewed and their difficulties discussed. In section 3.4 we examine the memory allocation problem through the concept of Working Set. This area has attracted considerable interest and intense efforts ever since its inception. But we feel that some of the questions involved are still unresolved. Finally, in section 3.5 the whole question of system performances will be carefully re-examined.

3.2 Concurrent Processes

Process is an abstraction of the activity of a processing unit. The precise moment of its inception is somewhat vague. It appears that the concept was the outgrowth of several major system design projects such as MULTICS and OS/360. It is, on a conceptual level, the smallest unit of work (of the individual user) that is contending for system resources. In OS/360 terminology, it is called a task. This abstraction was needed to aid in the implementation of the mechanism of preempting and restarting programs without affecting their computational results. No precise definition of a process has been agreed upon but the concept is generally accepted. It is to be distinguished from a program in that a program can be thought of as a collection of individual instructions occupying some address space, usually contiguously, therefore static, while the dynamic nature of the process is implicit. Then is seems quite natural to look upon an individual program as a collection of different processes. Processes arise and dissolve in the course of program execution, acquiring and releasing resources along the way. In other words, the unit of decomposition of a user's program is the process rather than the usual main program, subroutine(s) division which is a mere static partition in address space. The system is then a collection of different processes *from different users*. This view, in fact, pervades the design of THE multi-programming system [4].

A large number of processes in a system are independent from each other. Assuming the availability of resources, each such process is logically capable to proceed and in asynchronous manner. The example that immediately comes to mind is the CPU and channel overlap. There is no doubt that utilizing the concurrency in processes results in speed-ups. But due to its asynchronous nature, each process potentially proceeds at a different speed. The system's controlling mechanism then has the burden of coordination. The basic types of coordination needs are: 1) Synchronization; 2) Mutual Exclusion. There is another problem area due to limited resources among the competing processes, i.e. the deadlock problem.

If system resources are unlimited, there are only two states that a process can assume between its creation and termination, namely, active and blocked. A process in active state is a process that is productive. If, for some reason, the process cannot proceed any further, it is put in limbo, or, blocked state. Since it is assumed that unlimited amount of resources are

available, the reason for suspension is not for lack of processor or memory, etc. The process is blocked because other processes are "behind in progress"; this is the characteristic of the asynchronism among processes. The states are depicted in Figure 3.2.1.

Realistically, no system has unlimited resources. Therefore, there is a ready state in which a process is waiting for its resource requirements to be fulfilled. The situation is represented in Figure 3.2.2.

Consider a set of processes $\{p_1,p_2,\ldots,p_n\}$ each of which is in some state of execution and let $\{r_1,r_2,\ldots,r_m\}$ be a set of resources already in use by these processes. A process, p_i, which possesses r_j at the time and also wishes to acquire r_k can be depicted by a resource state graph [47] as in Figure 3.2.3.

Each resource is represented by a node. The label on the node is the process which currently possesses this particular resource. If this process is requesting additional resource, then it is represented by an arc from the current resource node to the resource node being requested. Hence, the resource state graph represents the state of current resource possessions and current resource requests.

For a simple 2 processes and three resources system, Figure 3.2.4 is a possible resource state graph. It shows that process p_1 is active, possessing r_1, and is requesting r_2; p_2 is active, possessing r_3, and is requesting r_1 and r_2. The situation presents no problem; p_2 will be blocked, assuming no preemption of resource r_1 from p_1. However, if there were a third process, p_3, already in control of r_2 and requesting r_3, then this would have created a circuit (directed loop) in the resource state graph as in Figure 3.2.5.

In this case, all three processes will be blocked and all three resources will not be available to the system pool. If there is no more other process either active or in ready queue, the entire system comes to a standstill, i.e., deadlock. It has been shown that the existence of a loop in the resource state graph, i.e., a circular wait for the processes, is a necessary and sufficient condition to cause all the processes involved to be blocked indefinitely if no outside intervention [10,11]. Clearly, the minimum number of active processes is two and the minimum number of resources is also two for a minimum loop to exist in a resource state graph.

Let us consider under what conditions that would lead to the circular circuit situation. They are as follows:

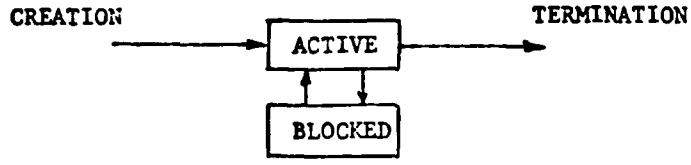


Fig. 3.2.1 Process States - unlimited resources

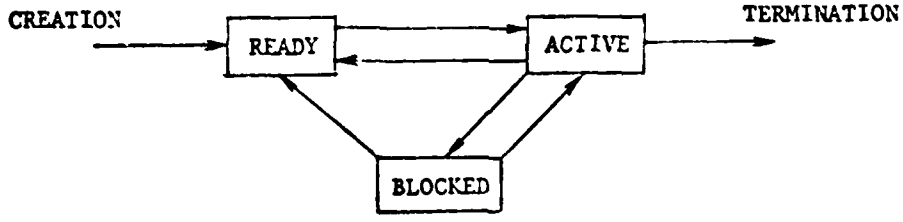


Fig. 3.2.2 Process States - limited resources

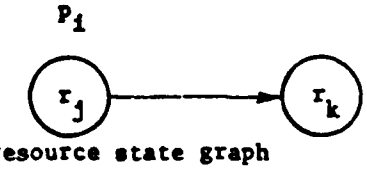


Fig. 3.2.3 A resource state graph

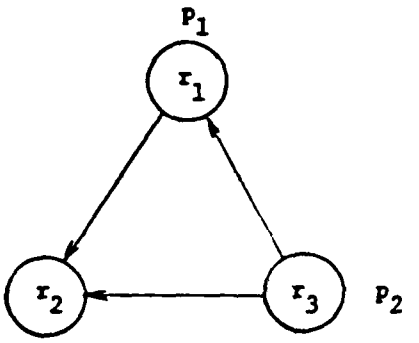


Fig. 3.2.4 Example of 3 resources state graph with 2 processes active

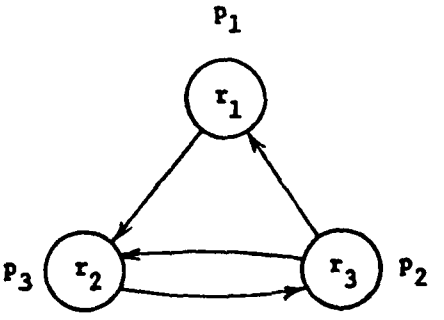


Fig. 3.2.5 State graph with a directed loop

1) Mutual exclusion -- processes claim exclusive control of the resources they require.
2) Wait for condition -- processes hold resources and asking for more.
3) No preemption -- resources cannot be forcibly removed.

In fact, in order to have a circular wait, all three must exist at the same time. Clearly, this presents some obvious strategy to prevent the deadlock from happening, i.e., to prevent the existence of one or more of the above conditions. We must allow each process to hold at least one of its resources exclusively, therefore, condition 1 cannot be denied. To allow preemption on some resources could be very costly, in terms of the potential waste on the processing already done. This leaves condition 2 which can be avoided by stipulating that each process must request all resources needed at once and cannot proceed until all have been granted. The latter approach had, in fact, been in use in some systems. Another approach is to impose a linear ordering of resource types on all processes. It can be shown that this hierarchical structuring would prevent the loop from existing in the resource state graph and this was the approach taken in the design of OS/360 MVT system [11]. The other ways of handling this deadlock problem are: detection and recovery and avoidance. These two are much more involved and are still under conceptual investigation. In any case, the deadlock problem exists as a logical possibility which arises from process concurrency and the way processes are managed. However, deadlock situation in real life is not prevalent. Whether or not one should have the costly prevention mechanism, or even costlier detection/avoidance algorithm built in is a matter of system design philosophy that is open to conjecture. The point is that such a situation is of low frequency in nature and its effect is not fatal in the sense of total loss in information, or worse, incorrect computational results. Which brings back our discussion on the question of correctness.

Since we envision the system as a set of asynchronously cooperating processes, it is imperative that the final result should be independent of the relative speed of individual processes. The contribution by Dijkstra in this regard is, of course, fundamental. He proposed a set of indivisible operations P and V, based on the concept of semaphore which by definition is non-negative on initiation [4].

These synchronizing primitives are used for the purpose of interprocess communication. A famous example was the so-called Reader/Writer problem, used

to illustrate the I/O activity. Not only do these primitives solve the synchronizing problem but they handle the mutual exclusion problem as well. These primitives have been used in various situations and assumed perhaps different names, e.g., LOCK/UNLOCK, WAIT/SIGNAL. It should be observed that the idea relies upon certain assumptions: that P and V can be operated without division and interruption; therefore, while in progress, it must be guaranteed exclusive access to any resources it requires. It should be remembered that this is a concept and a recipe; a concept that shows what is minimally required to achieve the synchronization (or mutual exclusion) requirement and a way to do it. In reality, it still depends on the actual hardware on hand and the method of implementation. This is not so straightforward [12]. Furthermore, it is up to the individual designer to provide a queueing discipline to handle the queue of the blocked processes. Also, provision must be made to preclude the possibility of a process being blocked indefinitely.

Since its inception, the concept of semaphore has been much refined and capabilities added. It is not our intention to give a complete treatise on the subject in our current discussion. A comprehensive treatment can be found in a book by Brinch Hansen [48]. In summing up this portion of the discussion, we will make the following two observations:

1) Perhaps the most important point of the P/V operations is that in thinking this way, a programmer forces himself to use a set of mental tools and when employed carefully, the resultant codes are at least demonstrably correct.
2) The questions of resource utilization, system efficiency, and the speed with which the processes proceed, hence the overall system throughput, do not come into the picture.

3.3 Queueing Model

3.3.1 Preliminary

Queueing theory is one branch of applied mathematics, i.e., applied probability, and is also known under other names such as traffic theory, congestion theory, the theory of mass service, and the theory of stochastic service systems, etc. It has been developed in an attempt to predict fluctuating demands from observational data and to enable an enterprise to provide adequate service for its customers with tolerable waiting time. The first

analytical treatment of Poisson queues was applied to the telephone traffic engineering by A. K. Erlang of Copenhagen.

In a service facility, queues (waiting lines) are formed due to the scarcity of resources. Many familiar situations arise in many diverse segments of real life. For example: the relationship between the in-patients, out-patients and the number of sick beds in a hospital; aircraft landing and take-off rates and the number of runways in an airport; arrival and departure of ships and the docking (loading/unloading) facilities of a harbor; traffic patterns in the city, toll booths, tunnel, etc.; inventory in business; repairmen problem; assembly line problem, so on and so forth. Queueing theory has been applied successfully to various situations in all of the above categories when the various probability distributions were carefully observed and the model being carefully constructed and analyzed. It was successful in the sense that the prediction (expectation value) agreed quite well with the actual value (observational data). Many illuminating examples were given by Saaty [13].

Let us now examine the situation concerning the computer systems. From the utility point of view, users are the customers, the system is the facility. Resources (CPU, memory space, I/O channels, etc.) are scarce. Queues are necessarily formed. Given the successes of queueing theory in other congestion problems as a back drop, it is quite natural to try a similar approach to computer systems. In the ensuing discussions, we will first present the basic hypotheses and results of a simple queue. Then, queueing in the context of computer systems, both theoretical and systems modelling are briefly described and then a re-examination on some problem areas.

3.3.2 A Basic Queue

A queueing model consists of servers where customers receive services and of queues where customers can wait if all servers are busy. The simplest model has one server with one queue, a single channel queueing model, as in Figure 3.3.1.

A queueing model with above structure is completely characterized by three things: input process, service distribution and queueing discipline. A set of differential-difference equations governing the interrelationship among the members of the queue can be derived from birth-death process and the associated hypotheses. See, for example, Cooper [14], and of course, Feller [15].

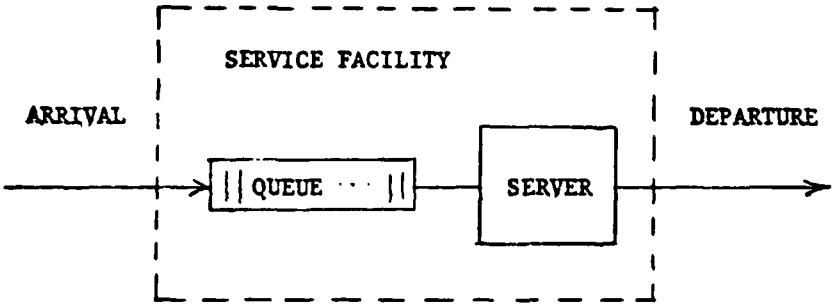


Fig. 3.3.1 Single Channel Queue

A. Birth and Death Process

Consider that, for each fixed time t, $0 < t < \infty$, a system can be described by a random variable N(t) which assumes a certain value with a probability. There is no loss of generality if we consider integer values. We say that a system is in state E_j at time t if $N(t) = j$, $j = 0, 1, 2, \ldots$

Postulates:

(1) System only changes through transitions from states to their next neighbors, i.e.

$$E_j \to E_{j+1} \quad \text{or} \quad E_j \to E_{j-1} \quad \text{for} \quad j \geq 1$$

and $E_0 \to E_1$ only.

(2) If at time t the system is in state E_j, the conditional probability that during the interval (t, t+h) the transition $E_j \to E_{j+1}$ occurs is equal to $\lambda_j h + 0(h)$, $j = 0, 1, 2, \ldots$ λ_j is the birth rate of the system at state E_j and 0(h) denotes a quantity which is of smaller order of magnitude than h. That is,

$$\lim_{h \to 0} \frac{0(h)}{h} = 0$$

(3) Similarly, the conditional probability of the transition $E_j \to E_{j-1}$ is equal to $\mu_j h + 0(h)$, $j = 1, 2, 3, \ldots$

(4) The probability that more than one change occurs during interval (t, t+h) is 0(h).

A process obeying the above postulates is called a birth-and-death process.

If we denote $P_j(t)$ to be the probability that the system is in state E_j at time t, i.e., $P_j(t) = P[N(t)=j]$, then we can calculate $P_j(t+h)$ from the above postulates. At time t+h the system can be in state E_j with the following possibilities:

(1) At time t, system is in E_j, during (t, t+h) system stays in E_j.

(2) At time t, system is in E_{j-1} and that transition $E_{j-1} \to E_j$ occurs during (t, t+h).

(3) At time t, system is in E_{j+1} and that transition $E_{j+1} \to E_j$ occurs during (t, t+h).

(4) More than two transitions occur.

It follows from the postulates and the law of total probability that

$$P_j(t+h) = [1-(\lambda_j+\mu_j)h]\, P_j(t) + \lambda_{j-1}hP_{j-1}(t) + \mu_{j+1}hP_{j+1}(t) + 0(h)$$

$$\frac{P_j(t+h) - P_j(t)}{h} = -(\lambda_j+\mu_j)P_j(t) + \lambda_{j-1}P_{j-1}(t) + \mu_{j+1}P_{j+1}(t) + \frac{0(h)}{h}$$

Note that

$$\lim_{h\to 0} \frac{0(h)}{h} = 0 \quad \text{and} \quad \lim_{h\to 0} \frac{P_j(t+h) - P_j(t)}{h} = \frac{d}{dt}\,[P_j(t)].$$

We have

$$\frac{d}{dt}\,[P_j(t)] = \lambda_{j-1}P_{j-1}(t) + \mu_{j+1}P_{j+1}(t) - (\lambda_j+\mu_j)P_j(t) \tag{3.3.1}$$

subject to

$$\lambda_{-1} = \mu_0 = P_{-1}(t) = 0. \tag{3.3.2}$$

Furthermore, if at $t = 0$ the system is in state E_i, the initial conditions are:

$$P_j(0) = \begin{cases} 1 & \text{if } j = i \\ 0 & j \neq 1 \end{cases}$$

The coefficients $\{\lambda_j\}$ and $\{\mu_j\}$ are called the birth and death rates, respectively.

B. Input (arrival) process of a queue - Pure birth

Pure birth means $\{\mu_j\} = 0$, and we shall consider the case of constant birth rate, i.e., $\lambda_j = \lambda$.

Assuming that system is in E_0 at $t = 0$, then

$$\frac{d}{dt}\,[P_j(t)] = \lambda P_{j-1}(t) - \lambda P_j(t) \tag{3.3.4}$$

and

$$P_j(0) = \begin{cases} 1 & j = 0 \quad j = 0, 1, 2, \ldots \\ 0 & j \neq 0 \end{cases}$$

$$P_{-1}(t) = 0$$

The general solution can be obtained recursively as

$$P_j(t) = \frac{(\lambda t)^j}{j!}\, e^{-\lambda t} \quad j = 0, 1, 2, \ldots \tag{3.3.5}$$

Note that

$$\sum_{j=0}^{\infty} P_j(t) = 1 \quad \text{for } t \geq 0 .$$

Recall that $P_j(t) = P[N(t)=j]$, we therefore conclude that the random variable N(t) has the Poisson Distribution with mean λt. In other words, we say that the arrival process for customers to a queue facility is Poisson.

C. Service Time Distribution

Consider a basic model of Bernoulli Trials. A Trial (experiment) is performed every unit time. A Failure is that the service continues and a Success means termination. Let p be the probability of success. Then $(1-p)^k$ is the probability of k trials resulting all in failures. In our context, this means that the waiting (or service time) is at least k since a trial is performed every unit time, i.e.

$$Pr[T>k] = (1-p)^k. \tag{3.3.6}$$

The expected number of success (terminations) in k trials is (kp). If k trials took place in interval t, then the mean number of success per unit time, μ, is

$$\mu = \frac{kp}{t} \tag{3.3.7}$$

Suppose $k \to \infty$, $p \to 0$ in such a way that

$$kp = \mu t = a > 0,$$

then

$$\lim_{\substack{p\to 0 \\ k\to\infty}} (1-p)^k = \lim_{k\to\infty} (1 - \frac{\mu t}{k})^k = e^{-\mu t} \tag{3.3.8}$$

Therefore, Bernoulli Trials lead to Geometric Distribution and in the limit it approaches the Negative Exponential Distribution. Furthermore,

$$Pr[T>t+\Delta t \mid T>t] = \frac{Pr[T>t+\Delta t]}{Pr[T>t]} = \frac{e^{-\mu(t+\Delta t)}}{e^{-\mu t}} = e^{-\mu(\Delta t)} = Pr[T>\Delta t].$$

This means that the probability for a customer to wait (being served) additional time, Δt, is the same regardless how long he has waited. This is the Markovian property (memoryless).

D. A Stationary Queue

For statistically equilibrium case, $P'_j(t) = 0$ for all j. Then, Eq. (3.3.1) becomes

$$\mu P_{j+1} + \lambda P_{j-1} - (\lambda+\mu)P_j = 0 \quad j > 0 \tag{3.3.9}$$

and

$$\mu P_1 - \lambda P_0 = 0 \tag{3.3.10}$$

subject to

$$P_{-1} = 0, \quad \mu P_0 = 0 \tag{3.3.11}$$

The solutions are straightforwardly obtained as:

$$\begin{aligned} P_j &= \left(\frac{\lambda}{\mu}\right)^j P_0 \\ &= \rho^j P_0 \quad j = 1, 2, 3, \ldots \end{aligned} \tag{3.3.12}$$

where $\rho = \frac{\lambda}{\mu}$ is called load or utilization factor.

E. Example of a Single Server Facility with Finite Queue Length

Consider a facility with a capacity for N customers. There is a finite number of states, i.e. 0 to N. The finite capacity of the queue is (N-1) since one customer is being served. The system of equations are:

$$\begin{aligned} &\mu P_1 - \lambda P_0 = 0 \\ &\mu P_{j+1} + \lambda P_{j-1} - (\lambda+\mu)P_j = 0 \quad 0 < j \leq N-1 \\ &\mu P_N - \lambda P_{N-1} = 0 \\ &P_{-1} = P_{N+1} = 0 \\ &\mu P_0 = \lambda P_N = 0 \end{aligned} \tag{3.3.13}$$

Clearly, $P_n = \rho^n P_0$ for $0 \leq n \leq N$. Since P_i's are Probability functions, the normalization condition must hold, i.e.

$$\sum_{i=0}^{N} P_i = 1 = (1+\rho+\rho^2+\ldots+\rho^N)P_0$$

$$= \left(\frac{1}{1-\rho} - \frac{\rho^{N+1}}{1-\rho}\right)P_0$$

$$= \left(\frac{1-\rho^{N+1}}{1-\rho}\right)P_0$$

Hence

$$P_0 = \frac{1-\rho}{1-\rho^{N+1}}$$

$$P_i = \left(\frac{1-\rho}{1-\rho^{N+1}}\right)\rho^i \quad i = 1, 2, \ldots, N \tag{3.3.14}$$

The mean number (expected value) of customers in the system (in service and in queue) is

$$\bar{L} = \sum_{i=0}^{N} iP_i$$

$$= P_1 + 2P_2 + 3P_3 + \ldots + NP_N$$

$$= \rho(1+2\rho+3\rho^2P\ldots+N\rho^{N-1})\left(\frac{1-\rho}{1-\rho^{N+1}}\right)$$

Note that

$$1 + 2\rho + 3\rho^2 + \ldots + N\rho^{N-1} = \frac{1-(N+1)\rho^N+N\rho^{N+1}}{(1-\rho)^2}$$

Therefore, we obtain

$$\bar{L} = \rho\left[\frac{1-(N+1)\rho^N+N\rho^{N+1}}{(1-\rho)(1-\rho^{N+1})}\right] \tag{3.3.15}$$

It can be shown that $\bar{L}$ has the following expressions for various approximations on , i.e.

$$\bar{L} \simeq \begin{cases} \rho + \rho^2 & \rho \ll 1 \\ \frac{N}{2} + \frac{1}{12}N(N+2)(\rho-1) & \rho \to 1 \\ N - \left(\frac{1}{\rho}\right) & \rho \gg 1 \end{cases} \tag{3.3.16}$$

The mean square fluctuation $(\Delta\bar{L})^2$ can also be shown as

$$(\Delta\bar{L})^2 = \sum_{i=0}^{N} i^2 P_i - (\bar{L})^2$$

$$= \frac{\rho - (N+1)^2 \rho^{N+1}(1-\rho)^2 - 2\rho^{N+2} + \rho^{2N+3}}{(1-\rho)^2(1-\rho^{N+1})^2}$$

$$\simeq \begin{cases} \rho + 2\rho^2 & \rho << 1 \\ \frac{1}{12} N(N+2) & \rho \to 1 \\ \frac{1}{\rho} + \frac{2}{\rho 2} & \rho >> 1 \end{cases} \qquad (3.3.17)$$

The mean number in queue is

$$\bar{L}_q = \sum_{i=1}^{N} (i-1) P_i$$

$$= P_2 + 2P_3 + 3P_4 + \ldots + (N-1)P_N$$

$$= \rho^2 [1 + 2\rho + 3\rho^2 + \ldots + (N-1)\rho^{N-2}] P_0$$

$$= \frac{\rho^2 [1 - N\rho^{N-1} + (N-1)\rho^N]}{(1-\rho)(1-\rho^{N+1})}$$

$$\simeq \begin{cases} \rho + 2\rho^2 \\ \frac{1}{2}(N-1) + \frac{1}{12}(N-1)(N+7)(\rho-1) & \rho \to 1 \\ N - \frac{1}{\rho} - 1 \end{cases}$$

F. Interpretations and Discussion

(1) For $\rho << 1$, i.e. service rate is a lot higher than the arrival requests, then

(i) $\bar{L}$ is small.

(ii) $\Delta\bar{L} \simeq \sqrt{\rho}$

$$\frac{\Delta\bar{L}}{\bar{L}} \simeq \frac{\sqrt{\rho}}{\rho} > 1$$

In other words, the variance, or the deviation from the mean, is large when compared to the mean itself.

(iii) P_0 is the probability of the system being in state of 0, i.e., no customer at all,

$$P_0 = \frac{1-\rho}{1-\rho^{N+1}}$$

$$\simeq 1$$

therefore, the system is idle a large percentage of the time.

(iv) P_N is the probability that the system is full. If we agree that when system is full, newly arrived customers will be turned away, therefore, "lost".

$$P_N = \rho^N P_0$$

$$\simeq \rho^N << 1$$

This is to say that when a system has a fast service compared to arrival requests, waiting line is very unlikely to reach its capacity therefore very unlikely to lose any customers.

(2) For $\rho >> 1$, i.e. average service time is a lot longer as compared to the average interval of requests.

(i) $\bar{L} \simeq N$

(ii) $(\Delta\bar{L}) \simeq \frac{1.4}{\rho}$ which is small if $\rho >> 1$.

(iii) $P_0 \to 0$ and $P_N \to 1$

(iv) All the above point to the conclusion that the system is full most of the time and consequently a large number of customers will be lost.

(3) For $\rho \to 1$, i.e. the load is nearly balanced.

(i) $\bar{L} \to N/2$ if $\rho = 1$.

However, the expected number of customers in system can be quite sensitive to the load factor ρ, if N is large.

(ii) $$P_0 = [\frac{1-\rho}{1-\rho^{N+1}}]$$

$$= \frac{1}{1+\rho+\rho^2+\ldots+\rho^N}$$

$$= \frac{1}{N+1}$$

$$P_N = [\frac{1-\rho}{1-\rho^{N+1}}]^N$$

$$\simeq \frac{1}{N+1}$$

In other words, when $\rho \to 1$, the fraction of time when system is either idle or full is in inverse proportion to the system capacity.

(iii) $(\Delta\bar{L}) \simeq \frac{N}{12}$, if $N >> 2$, which is quite large if N is large.

(iv) Therefore, we may conclude that although the number of customers in the system varies greatly, the system is mostly busy and customers are unlikely to be turned away.

Summing up, cases (1) and (2) are certainly undesirable. In case (3), we may have some room to work with. There are several approaches to tuning the system:

(i) For a fixed N, we would adjust P, the load factor, to be either a little over or under unity so that the expected number, $\bar{L}$, hence the average waiting time for a newly arrived customer is acceptable according to a certain criteria.

(ii) For a given ρ, $\rho \to 1$, we could choose N such that P_0 and P_N come within a certain predetermined value.

However, large N means higher cost and for a given customer population, λ is fixed. Hence μ is the other area a designer may be able to exercise some control. In any case, the freedom of choice is quite limited.

3.3.3. Queueing in the Context of Computer Systems

Look upon as a whole, the computer system is the "server" and the individual programs submitted by users are the "customers". This forms a single channel, one server queue. But this all embracing model necessarily buries a lot of details. If we look closely, we would find processor queue, memory queue, I/O queue, etc. In fact, all the scarce resources form their own queues. However, in each of these queues, the interpretation of incoming "customers" and "arrival processes" are more subtle, and the interpretation of service mechanism, i.e. service time distribution is very difficult. If a system supports both multiprogramming batch mode and time-sharing interactive mode, its queue model would take on an extra degree of complexity. It should be noted that multiprogramming need not be time-sharing and that a time-sharing system is not necessarily multiprogrammed. For example, the CTSS system of MIT is a time-sharing system with at most one user in core

in any given time. Nevertheless, time-sharing system not only shared processor among users, as does the multiprogram system, but also further characterized by interactive (conversational) accesses. We may also look upon a time-sharing system with multiple users in core as that of memory sharing as well, though not in the time-multiplex sense as the processor sharing. In general, a majority of the theoretical treatments on queueing models are time-sharing queues. Moreover, they are of processor queues.

In a time-sharing environment, a user is sharing the server (processor) in a time-multiplex fashion. But in a given user-system interaction, the required processor services may exceed one time allotment. Therefore, it is natural to think of the user as being ejected from the server (processor) and then rejoin at the end of the queue. Instead of the first-come-first-served discipline as the simple queue discussed in last section, we have now a modified queue discipline (or service discipline) for the time-sharing queue. Generally, they can be classified as Round-Robin (RR) or Priority or Foreground-Background (FB) Queue. (Fig. 3.3.2)

Kleinrock [16] was among the first to study this model. Rasch [17] further studied this RR model, with and without overhead. Priorities were also assigned to incoming users based on decreasing function of program length. Heacox and Purdom [18] continued to investigate the variation of RR model. They called their model SQ (Single Quantum) and MQ (Multiple Quantum) Model. SQ is the conventional RR model. However, MQ is a modification in which the amount of service per pass depends on the rate at which programs arrive in the system.

Many people have studied the N-Level Foreground-Background (FB_N) Model (Fig. 3.3.3.). Among those are Coffman and Kleinrock [19]. All but level 1 are background queues. The distribution of waiting time is controlled through the altering of the processor time quantum size or changing the number of levels, N. Additional control may be obtained through the use of external priority assignments. It should be noted that both FB_N and RR models are all variants of a class of feedback queues. Voluminous amounts of literatures exist in the area of analytic models of time-sharing. A good overview in this aspect can be found in [20].

By and large, the types of analytic time-sharing models are limited and the models differ only in the choice for the following parameters:

(1) Number of input channels (finite or infinite)

(2) Number of processors

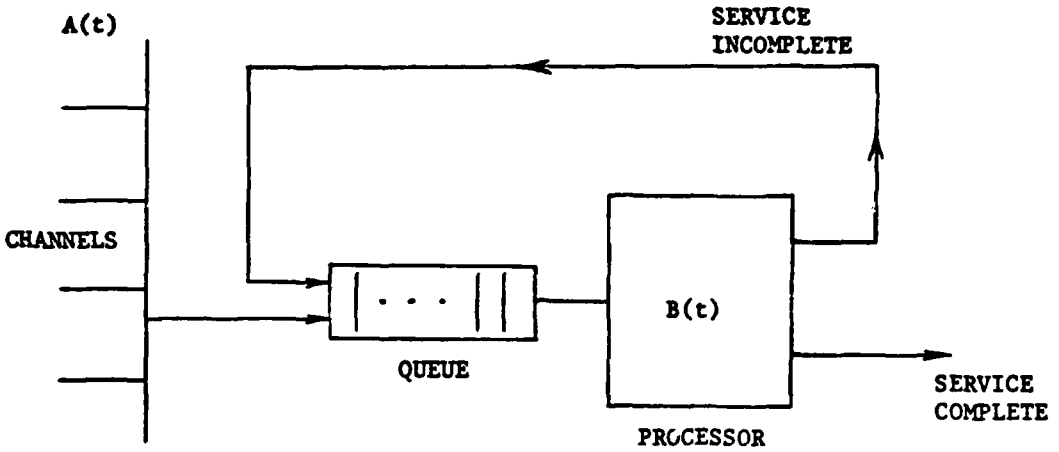


Fig. 3.3.2 Round-Robin (RR) Model

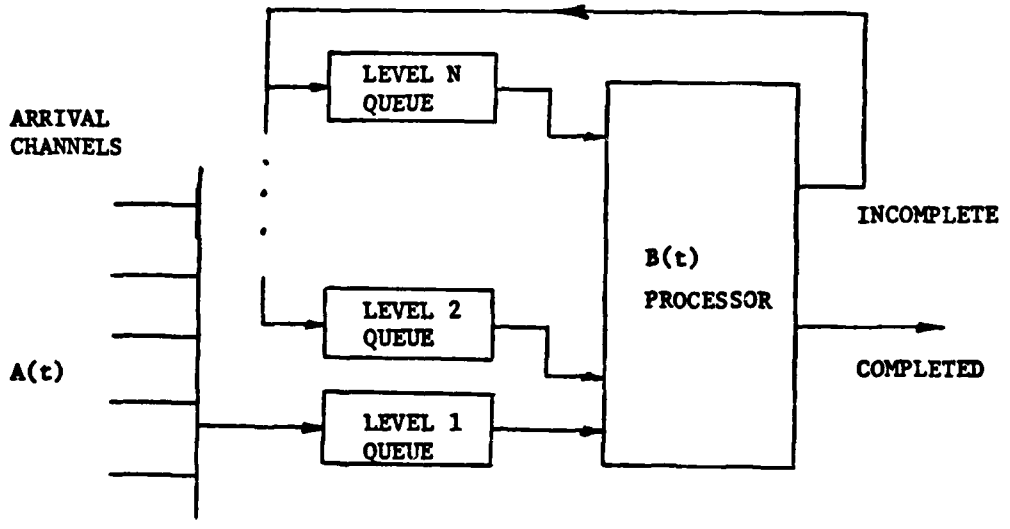


Fig. 3.3.3 N-Level Foreground-Background (FB_N) Model

(3) Arrival processes, e.g. Bernoulli, single Poisson stream, multiple Poisson stream

(4) Service distributions, e.g. Geometric, Exponential

(5) Handling of swap time T_s and overhead time t_0, e.g. $T_s = T_0 = 0$, $T_s + T_0 = T$ = Constant or T = random

(6) Quantum assumption, e.g. finite processor quantum or processor-sharing (infinitesimal processor quantum).

In the case of multiprogramming systems, queueing studies appear to be relatively few; certainly not comparable to the time-sharing situation. Mitrani [21] studied a simple cyclic queue model of a multiprogramming system with a fixed number of tasks. Specifically, the system consists of two servers, namely, a processor (CPU) and a channel (PPU) in series and each task must cycle through these two servers a number of times (random) before termination. Kleinrock [22] also treated a model with many processors in series. Shedler [23] considered a cyclic queue with the processor in series with two levels of memory system in which there is sequential dependency of accesses between the devices.

In actual system studies, perhaps one of the most well known efforts was the pioneering work by Scherr [24] on CTSS (Compatible Time-Sharing System) at MIT. The stated purpose of his work was simply to understand the various factors involved in an interactive, time-sharing environment. Anderson and Sargent [25] analyzed an APL/360 system. Moore [26] modelled MTS (Michigan Time-Sharing) system with a network of queues. Each had various degrees of success and we will discuss some of them, as well as the areas of difficulties.

3.3.4 Discussions

By studying a very simple and basic queue model, we see how some of the basic parameters can be obtained. But it is also quite clear that in attempting to adjust some of these parameters to accommodate the situation better, i.e. less waiting or congestion, the choice is rather limited. We have also seen the proliferation of analytic queueing models. In the treatment of time-sharing case, it is invariably the processor queue with a feedback structure. And the multiprogrammed system is represented by CPU-I/O cyclic queue. For mathematical tractability, the arrival process and service time distribution are mostly Poisson and Exponential, sometimes Geometric.

In the case of time-sharing, there were experimental evideuces to support the assumption that the inter-arrival distribution is nearly Poisson [25, 27,28]. However, the service time, i.e. processor busy time, is not Exponential. In fact, it is not even Hyperexponential [28,29]. In a system with limited number of terminals which is always the case, then each arrival, i.e. a user request, is not entirely independent from all others. This is reflected on the fact that when the system load is heavy, response time will be longer and then each user will have to wait longer until the initiation of next request. Consequently, the input rate becomes a function of the service rate. One might further question the validity of characterizing the entire system with a processor queue alone. Finally, the stationarity, i.e. statistically equilibrium, is always assumed but seldom verified. While it is reasonable to use the mean response time to a terminal user as the measure of time-sharing system performance, it is not clear how one might effectively improve such figures by controlling the processor time quanta. For if we characterize a system by a processor queue, so many other factors are lumped together in the guise of Service Time, often exponentially distributed, and not accurate at that. For example, how does one represent the delay in service completion due to page fault? Since the page replacement policies are built into the controlling routines of the operating system, changing the quantum size of processor time would seem futile in trying to affect an improvement in response time. In dealing with multi-programmed batch users, other variables such as the programming language used, individual coding styles, other system resources, etc. would take on more significance in the question of overall performance. Finally, when a system serves a mixture of batch and time-shared users, finding an analytic model seems out of reach. Simulation does appear to offer promising results but it requires a careful construction of model that closely resembles the real system. Scherr [24] obtained simulated results that agreed well with observations. And he took pain to point out that the result represented CTSS and CTSS-like system only. Anderson and Sargent [25] also had good results in their studies. Perhaps it is worth noting that the former case involved in a study of a constant user pool and the system was dedicated to time-sharing. It was found that only mean think time, mean processor time (including swapping) and the number of users interacting with the system are of first order effect. On the other hand, Anderson and Sargent [25] found that the quantum size was a most important factor in performance, in terms of response time;

they were studying an APL/360 system. Moore [26] studied a system that is not as homogeneous as the above two, i.e. a system of mixed batch and time-sharing. He did construct a network of queues based on some of the system's most prominent resources. But in so doing, that a job, or a customer would go through the system being restricted by the model. In reality, the way a job might traverse the systems, i.e. migrating from resource queue to resource queue, depends upon the knowledge a user has on the system and also the operating system's allocation and scheduling policies. It is not clear how queueing models, even a network of them, might take care of this situation. Finally, we should recognize that queueing theory plays essentially the role of analysis. We may hope to understand the system under study a bit more if we are careful in model construction, namely, system tuning. But it is not so promising if we try to synthesize a system, based on queue models, so that a system would attend a certain level of operational characteristics. The lack of controlling means in a queue model for system designers to consider alternatives under different resource constraints makes this point apparent.

3.4 Working Set Model

A basic program property is that the dynamic footprints of a processor, i.e., the instruction addresses actually visited by the processor during execution, scatter nonuniformly over the uniform address space of the program. This is generally known as locality.

Principles of Locality (assuming paging machine)

(1) During any interval of time, a program distributes its references (addresses) nonuniformly over its pages

(2) Taken as a function of time, the frequency with which a given page is referenced tends to change slowly, i.e. it is quasi-stationary.

(3) Correlation between immediate past and immediate future reference patterns tends to be high, whereas the correlation between disjoint reference patterns tends to zero as the distance (in terms of time) between them tends to infinity.

Denning has generalized these observations into a user program model, i.e. the Working Set Model [30]. Specifically, the working set $W(t,\tau)$ of a process (program) at time t is the collection of distinct information items

referenced by the process during the process time (virtual time) interval $(t-\tau,t)$. Therefore, if we use page as a basic unit of information items, $W(t,\tau)$ will be the set of distinct pages being referenced τ time units past (with respect to t). τ is called the working set parameter. Clearly, for small τ, the size of $W(t,\tau)$ will be small and larger τ will result in more pages belonging to the current working set. Having thus so defined, the term and its associated concept can be used to discuss some automatic memory management problems. (Fig. 3.4.1)

Because of locality, the processor dwells on a certain portion of the program more often than the others. We have the notion that memory space is being effectively used if the processor spends most of its time in that portion of the space. Similarly, it is wasteful on the memory if the portion of a program which is only touched by the processor infrequently also resides in the memory the same length of time as the other more frequently "used" addressing space. However, some parts of the program may be "used" a bit more by the processor than others, but no parts of a program would be "used" always, in a given time interval. From the system's point of view, it is desirable to allocate the amount of memory space to a user program that is just enough to cover the range of processor references within a certain time interval (virtual time) so that there will be more space for other users while the space just allocated will be utilized quite efficiently. For the user's point of view, anything less than total program space in memory means that in a certain point in time, the information item that is needed is missing and a delay in the progress of his computation will thus result, until the needed information is found and loaded into memory. The less memory allocated to a user, the more delay it will be. Clearly, there must be a compromise. On a conceptual level, we may say that from programmer's view point, the working set is the smallest collection of information that must be present in main memory to assure efficient execution of his program. Consequently we may define that working set size is the largest amount of memory that the system is willing to allocate to the user so that the allocated memory space will be utilized efficiently. This remark is imprecise. Firstly, we don't know exactly what constitutes an "efficient usage". Secondly, we don't know the size of a working set even though we know its definition.

If a new page is brought in on demand, and if we do not allow individual

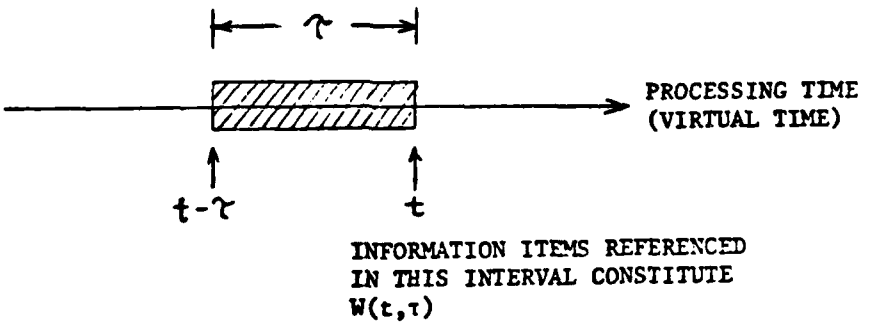


Fig. 3.4.1 Definition of W(t,τ)

user's own pages to grow indefinitely, at some point some pages will have to be displaced back to the secondary memory. Generally, the rules by which the system displaces the pages are known as replacement algorithm. There are many varieties such as random or least recently used (LRU). If we agree that a page may be displaced after it falls out of the working set, then the working set model actually constitutes a page replacement algorithm. By definition a page is out of the working set when it has not been referenced in τ time units. Since a page is brought in on demand, then it resides in memory at least for τ units of time. Assuming we know how to select τ, and also assume that τ be kept constant throughout the execution, the working set size is still a function of t and it may grow or shrink as the program property so dictates. If we call a replacement algorithm "perfect" in the sense that when a page is displaced, it never returns. In so doing, that page may be kept in memory unnecessarily long so that it may be referenced for the last time. Memory usage is certainly not high. On the other hand, a really bad choice in page replacement would cause the page to be returned "on demand" after it had just been displaced back to the secondary memory. A replacement algorithm based on the work set principle appears to be a compromise. However, how good a compromise hinges on the choice of τ, and this is still an open question. It has been observed that a phenomenon, called "thrashing", exists in a paging machine. Denning attributed [30] this to be the overcommitment of memory: when too many working sets occupy main memory, each is displacing another's pages in an attempt to have its own pages present. This view reflects the concept that the collapse of the system (thrashing) is due to too many users in memory. While it is true that for a given τ, the working set size is a function of t and it can grow, it cannot displace pages from other working sets if their pages had not been in memory for at least τ units of time, by definition. If a particular working set wants to grow (as demand arises) but found that all memory spaces were occupied and no page had been in residence longer than τ, this particular set, hence the program would simply be blocked. Instead, we feel that the cause of thrashing is the result of improper choice of τ in the case of the working set scheduler and a bad replacement algorithm in general, creating excessive return traffic, causing most of the processes in page wait state. In any case, working set is a concept by which memory management policy can spring forward. Denning proposed a "balanced" criterion [30]. It is basically an equipment utilization

criterion. In the context of working set, it is to load the individual working sets $W_i(t,\tau)$ such that

$$\sum_i |W_i| \leq \beta M \quad 0 < \beta \leq 1$$

where M is the total memory in pages, and β is predetermined. In a way, this is a "constrained" approach to resource (memory) allocation.

3.5 The Performance Question

System performance is one area of abstractions that is notably lacking. It is something that is always implicitly assumed in every area of studies in operating systems but rarely explicitly defined. Nor can this situation be amended readily. For it assumes a different meaning in different contexts. As a rule, present day operating systems are modularized horizontally with separate modules corresponding to distinct features of the operating system. But one of the great lessons about operating system performance is that a system is not well represented by the sum of its parts. The response of an operating system to various loads (resources demands) tends to be highly non-linear; the modules of the system strongly interact. We cannot even specify what the individual module's performance level should be, let alone the performance interaction. In fact, what do we consider as performance? A most often adopted point of view is that of time. Either we speak of speed (per time unit) or utilization (percentage of time). While these figures may be indicative of performance level in some pure, absolute sense, it may not be so meaningful in terms of reality. It is reasonable to set some ranges of terminal response time in a time-sharing system as a desirable performance level relative to the total terminal users, it cannot be used as a design specification for a time-sharing operating system. A layered, i.e., vertically modularized, operating system [4] has the virtue of clean interface between layers and that the correctness of each layer can be achieved if the functional and dynamical properties are well defined for each layer. This is in the pursuit of reliability (correctness) at the expense of some execution speed in the modules. Can one not agree that reliability is one form of performance also? There is no question on the degree of sophistication of today's hardware. It is equally clear that without proper supporting softwares, their potential can hardly be realized. If we view the entire system as a combined package to provide efficient computing

utilities to users, the user interaction cannot be ruled out of the picture in considering the question of performance. For example, how do we gauge the user satisfaction as a function of the total services that the system provides? A very flexible yet easy to use system control language, a comprehensive as well as comprehensible debugging package would mean more to the user at a program formulation stage than sheer speed. In this sense, turnaround time or system throughput, vague as they already are, would be of even less consequence. From a system's point of view, if we specify the maximum equipment utilization to be the performance index, not only this will clash with the user's own objectives, but it is also possible that a carelessly implemented policy toward this end could lead to adverse results, e.g., thrashing. Measuring the processor, I/O channel and various other system resource utilizations are some of the "common" approaches in attempting to evaluate the system. This is based on the old adage that performance automatically means speed and vice versa. While this may be meaningful to a certain extent, we feel that this view is too narrow. The question is whether a general viewpoint can be adopted as a goal so that when achieving it, we may say that the system is operating optimally. Conceptually, this goal is a reflection of our view toward the computer system in general. For instance, we may choose maximum processor usage or maximum throughput as our goal, no matter how dubious that may be. On the other hand, it is equally reasonable to set a goal for the system to give users with certain "qualifications" the maximum advantage of services. These "qualifications" can be the quantification of user program characteristics or other arbitrarily set standards. On the surface, this would seem that we have adopted an arbitrary measure for system performances by setting some arbitrary goals. But, in fact, the system goal viewed this way is just a generalization of the ordinary priority class concept. If the operating system is designed and implemented in such a way that it will always respond to and serve programs with the highest "qualifications" in the shortest possible time even at the expense of pre-empting other programs in execution and therefore resulting in a large amount of wasted (idle) resources, we would consider the system performance to be optimal in the sense that it accomplishes the predetermined goal. For we consider that if the system fails to serve the program with such qualifications, the "loss" could be much more detrimental than just some wasted resources. An example of this "absolute priority" type of program

already exists in the form of operator console routines. Generally, this "goal programming" approach will be reflected upon in the operating system's controlling modules. Specifically, the goal is accomplished through the system resource allocation and activity scheduling. We will come back to this question in Chapter 5 with specifics.

CHAPTER IV
Operating System Modelled as a Conglomerate of Inter-dependent Activities

4.1 Introduction

Operating system has been viewed as a set of cooperating sequential processes, having defined a process to be the smallest unit of work. Various systems of this kind have been built [4,6,8]. The main objective of modelling a system in this manner is to enable the study of interacting modules and the various requirements of synchronizing those processes. From this concept and the modelling efforts, one might be able to find some common principles with which the system functions can be partitioned into functional modules in a meaningful way. However, this approach does not entail the question of resource management in particular in system design. We have discussed earlier that we view the operating system functions as that of managerial nature and that the resource allocation question is paramount if we view the entire computer system to be an organization geared to production, i.e., computing utility. The ideas presented in this chapter are not unlike those of econometrics. Certainly, they are influenced by those of Dantzig [31]. Section 2 gives an overview on the activities of a representative multi-programming system and the analysis of the nature of those activities. Section 3 introduces the notion of mathematical programming and basic postulates on activities. A specific activity, namely, the allocation activity is formally described. This results in the common form of a linear programming problem. Section 4 briefly outlines the Simplex Method for solving the linear programming problem.

4.2 The Nature of the System Activities

The actions that are taking place within the computer system are extremely complex, to say the least. Different modules of the operating system respond to different events, both arising from user requests and the system's own operating status, in a generally unpredictable manner. Even though a certain action will take place in a certain sequence, most are random in the sense that the entire system's operation is based on interrupts, i.e., interrupt driven. In order to bring matters into sharper focus and to facilitate our discussion, a simplified overview of various

operating system activities is presented in Figure 4.2.1. This diagram does not represent all that is capable by a modern system but only that of a multiprogram environment, supporting local and remote batch job queues. Furthermore, the arrows do not reflect any hierarchical structure or precedence relations among the system activities. It is an indication of the kind of actions that a job (user program) may expect to experience as it flows through the system. Each circle indicates a group of activities for a specific purpose. These activities are carried out by various functional modules within the operating system. Certainly, they need their share of the resources in order to function properly. They all need memory spaces (resident routines) and some may need I/O channels, disc tracks, etc.; all will need processor time. Activities thus grouped is therefore the view as seen by a user program.

Input Media Conversion

When a job enters the system, either through on-line card reader or other input device (e.g., communications processor), it is separated into two files. One file contains the descriptions of the job (from job control cards) and the other is a data file which comprises various subfiles consisting of data to be used by the job. And then the job is on the system job queue.

Resource Allocator

Since a job (user program) may consist of one or more stages (job steps), resource allocation is given to each stage instead of the entire job. Memory and peripherals are assigned to a specific stage after the control tables (the control file built by the input media conversion activity) are examined to determine gross peripheral requirements. Through this technique of allocating peripherals in advance of execution, the operator can perform functions such as mounting tapes or specific disc packs while other program or system activities are in progress. It is during this allocation phase that the instructions are issued to the operator. When the assigned peripherals are ready and the memory management module finds enough space, this particular job step is put onto the ready queue. By implication, the ready queue consists of steps of jobs that are in memory. However, no step of a job is initiated unless all prior steps of that job have been completed

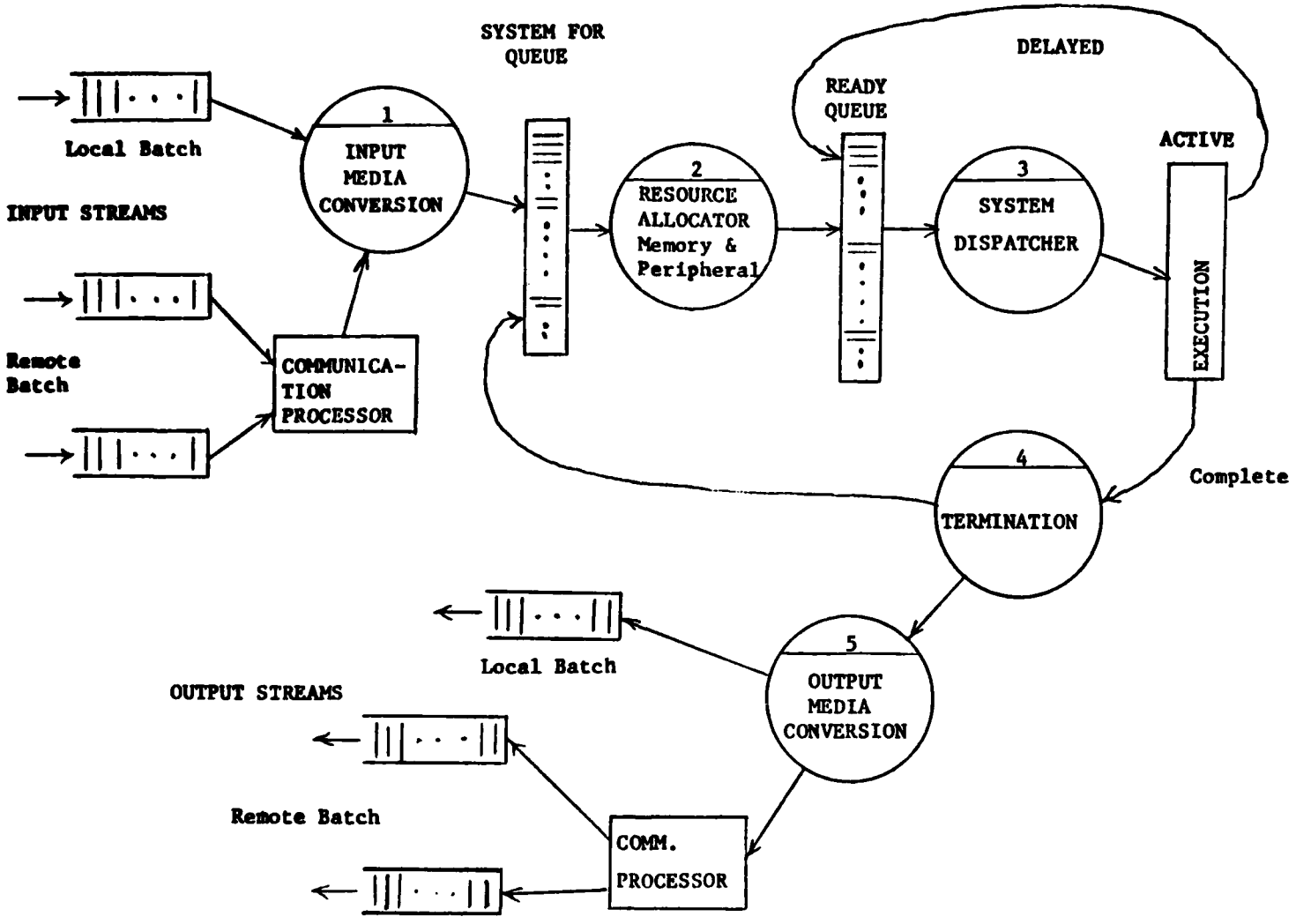


Fig. 4.2.1 Overview of operating system activities

or otherwise explicitly bypassed through job control cards under certain error conditions.

System Dispatcher

The dispatcher is the center of control for job execution. It only selects jobs from the ready queue, i.e., those jobs already residing in memory. To select a job to be executed is equivalent to saying that that particular job has been "given control" of the processor. A job that has the control of a processor is "active". It can be delayed due to either waiting for I/O completions or a forced relinquish by system timer interrupt. When it is delayed, it is put back onto ready queue, figuratively speaking, since the job was never physically removed from memory and is now being tagged for later resumption. The ready queue is dynamic in nature since it is interrupt oriented and since the dispatcher selects a job that would most readily utilize the processor rather than the relative position of jobs within the queue.

Termination

The job, or the particular step of a job, can terminate either normally or abnormally. Errors and accounting records are produced on the system output file, the operator is told if files require dismounting, tables used during execution are deleted, and memory and peripherals are deallocated. If the termination is only a job step and if there are more steps to come, then the next step in line which is on the system job queue will be marked for allocation consideration. If this is the final step, the system output file (for this job) is closed.

Output Media Conversion

All output (from different job steps) of a job are written on the system output file during execution, and is only transcribed into the output media (printer, card punch, etc.) after the completion of the entire job.

It is not explicitly shown but ever present in the system is the global event detector which is in essence the interrupt handler. It is this portion of the system activities that puts an active job back onto ready queue when delayed or decides that there are more job steps after each termination and that therefore the job flow should be directed back to resource allocator, etc.

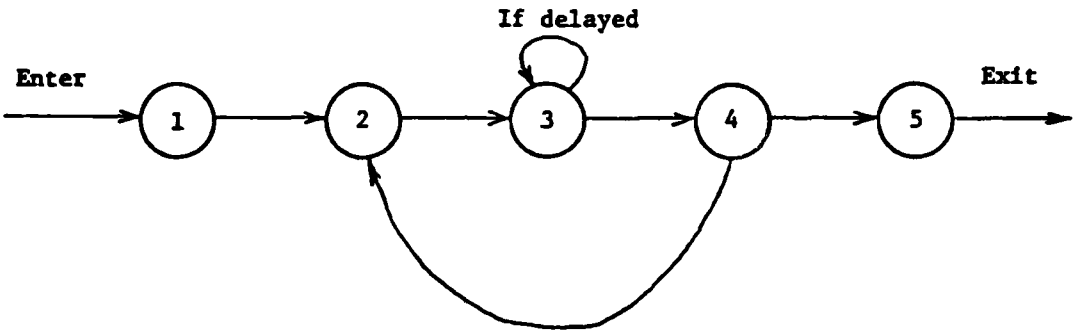


Fig. 4.2.2 Job flow example

Let us now consider a specific job. Suppose we want to process a program written in FORTRAN source statements together with data for execution if compilation is successful, a very common compile-load-go situation. From our previous discussion, this job comprises two stages (or steps). The first stage is compilation and the second stage is load-and-go. Note that in order to execute, the object module must be loaded first. Hence, load-and-go cannot really be considered as separable. In Figure 4.2.2 a simplified job flow diagram is presented. The number in each node corresponds to those in Figure 4.2.1. The activities so described are fairly typical of a multiprogrammed batch system. It should be pointed out that the five activity phases do not necessarily reflect the time-sharing environment. Furthermore, each program, or its steps, is executed in its entirety, i.e., non-paging, and that the only form of delay comes in processor preemption and there is no swapping (memory preemption) in the course of normal execution.

Resource allocation and scheduling are customarily considered as a single question. But as we have made clear in this overview that they are in fact two distinct activities within the system; scheduling only comes after the allocation phase. One may argue that in a broad sense, resource allocator actually "schedules" jobs from the system job queue to be executed by granting their requests for resources. Viewed this way, we feel the dynamic nature of job dispatching becomes obscured. Furthermore, these two activities may actually follow two quite different strategies each a result of the design decision of the operating system. As has been pointed out earlier, not only the user programs demand resources. A system activity, or group of system activities are necessarily supported by system resources. Sometimes, the activity itself requires file space (normally disc space since most operating systems are disc oriented) and I/O channels; this is on top of the necessary memory space for the routine itself and its share of the processor time to carry out the intended action. However, these resource requirements do not compete with the user programs in the sense that the resource allocator only deals with what is left to be granted among the competing users. If we look at all the activities as a whole, instead of the five different phases as viewed in the context of a job flow, their functional natures may be classified into the following three categories:

(1) Housekeeping: The activities that keep track of the system as well as the user program status. For example, the creation of

control files during input and deallocation of resources during output and various interrupt processings.

(2) Utility: Mainly, this concerns the supplying of all I/O routines to the users and the managing of all such I/O processings.

(3) Controlling: This category is manifested by the resource allocator and the system dispatch.

Regardless of how we may classify them, these activities are all interdependent in the sense that they either share or compete for limited resources and that there are also precedence relations among some of them. All of them exist for the expedient purpose of keeping a system operative and that a platform is provided for an individual user to accomplish his computational needs. In this light, the resource allocator and system dispatcher play a dominant role since they are the activities directly controlling the job flow. If we think of the entire computer system as a productive organization, then the operating system becomes the management which coordinates and supervises the various departments towards a common production goal. In this instance, user programs are the demands and the computing services thus rendered are the products. Clearly, in a limited resource situation, how best to dispense the available resources to achieve some predetermined production goal is the central issue. We have stated previously, in Chapter 3, that the performance question of a computer system gauged only in terms of time is too narrow a view. Now we are prepared to give a more general objective. That is: a system is optimal if it can achieve the chosen goal. And so the resource allocating activity should constitute a policy that is goal oriented. Even though it appears that the system production goal is wholly determined at the allocation stage, the dispatcher has its influence in an indirect way. Since the dispatcher strategy has bearing on how efficiently the various resources are being utilized <u>after</u> they are allocated to individual production demands, an efficient dispatcher can have an indirect effect on the overall production. In the next section we shall formalize the concept of optimal activity combinations under constraints.

4.3 Mathematical Programming and Activity Aggregates

4.3.1. The Programming of Activities

Programming, or program planning, may be defined as the construction of a schedule of actions by means of which an organization or other complex of

activities may move from one defined state to another, or from a defined state toward some specifically defined objective. Such a plan implies, and should explicitly prescribe, the resources and services utilized, consumed, or produced in the accomplishment of the programmed actions. Objectives must be stated in terms of basic ends thus permitting the consideration of alternative means, if they are to be useful in programming operations designed to optimize objectives within resource constraints. Mathematical programming is concerned with the problem of maximizing (or minimizing) functions under constraints. Its mathematical basis is mainly in the theory of linear inequalities and in the theory of convex sets and functions.

The notion of programming is a general one. On the user level, an individual program can be considered as the organization of activities, which, when successfully carried out, would achieve the objective of a computation. The trend of using high level language removes a user from the details of resource management. In fact, he is oblivious to them. On the system level, the main concern would be the proper coordination of individual user's activities under the limitation of system resources. It is the programming of the latter kind that we will be dealing with in the ensuing discussions.

4.3.2 Basic Assumptions

In the previous section, we have come to perceive the operating system as comprised of various observed activities. We may further contemplate that there exist some refinements as representative building blocks of different types that might be recombined in varying amounts to form yet more complex but possible activities. The whole set of possible activities will be referred to as a technology, i.e., the technology of operating systems.

Postulate I: There exists a set {P} of all possible activities.

Postulate II: There exists a finite set of basic activities, x_i, such that any possible state of an activity can be represented as

$$\sum_i a_i x_i \quad i = 1, 2, \ldots, n \tag{4.3.1}$$

where a_i is the level of basic activity x_i.

Postulate III: There exists a linear objective function,

$$\sum_i c_i x_i \quad i = 1, 2, \ldots, n \tag{4.3.2}$$

where c_i is a constant associated with x_i, depending upon a specific formulation of the objective function.

Furthermore, we shall assume that an activity in a possible state would consume (require) a certain amount of resources of a certain kind, possibly limited by a constant b, that is,

$$\sum_i a_i x_i \leq b \quad i = 1, 2, \ldots, n \tag{4.3.3}$$

4.3.3 The Allocation Activity

We will now discuss a specific activity, that is, the second group of actions as depicted in Figure 4.2.1., in the setting of those basic notions put forth in Section 4.3.2. The immediate task is to identify a finite set of basic activities. Since the resource allocator as outlined in Section 4.2 deals exclusively with user programs, it appears natural to choose the set of user programs on the system job queue as the set of basic activities. More precisely, since a user's program may consist of more than one stage (job step), it is that particular job step that is up for allocation consideration that becomes one component of this basis. If we further perceive that each possible state may consume different kinds of resources, and that if we adjoin these possible states together, we have

$$Ax \leq b \tag{4.3.4}$$

where A is a rectangular matrix, x and b will be column vectors, with orders compatible to each other so that (4.3.4) is meaningful. In reality, the number of elements in column vector b is equal to the total different resource requirements and other constraints by the set of basic activities, the column vector x. Thus, the allocation activity becomes the finding of a solution to (4.3.4). Since the feasible solutions to Eq. (4.3.4) are many, we may naturally want to find the most desirable one according to some criterion. We have thus come to the notion of goal oriented allocation. That is if we further establish a linear objective function in the form of Eq. (4.3.2) and set our goal to be the vector x that satisfies Eq. (4.3.4) and that also maximizes the objective function. This is stated formally as follows:

$$\begin{aligned} &\text{Maximize:} \quad \sum_i c_i x_i \quad i = 1, 2, \ldots, n \\ &\text{Subject to:} \quad Ax \leq b \end{aligned} \tag{4.3.5}$$

The vector x^0 that satisfies Eq.(4.3.5) is the optimal solution and an allocator which selects the individual jobs (or job steps) corresponding to the components of x^0 is said to have an optimal allocation policy.

4.4 The Simplex Method

4.4.1 Preliminary

Although a linear programming model for an economic problem had been developed as early as 1939 by the Russian mathematician L. Kantorovich, his work was not known outside the Soviet Union until 1955. Moreover, it did not lead to a mathematical theory or to numerical methods. Therefore, it is generally agreed that the year 1947 is considered as the origin of mathematical programming when George B. Dantzig developed the simplex method for linear programming [32]. Since then, it has certainly grown both in width and in depth. However, it is not the intention of this section to give a complete exposition on the theory of simplex method and linear programming. But rather, we will only present the method of solution, in the form of step-by-step procedures. Terminologies will be explained or when suitable, given in the form of definitions. Canonical forms of the Direct Problem (of linear programming) and its Dual will be displayed. An important Dual Theorem will be stated. No proofs will be given. In other words, the materials in this section serve as a vehicle for later discussions. Detail information is readily available from the references. In addition to the original work of Dantzig [33], excellent treatments can be found in [34,35].

4.4.2 Basic Definition

Definition 1: Let $A \neq 0$ be an n-component row vector and b a scalar; the set of all x for which $Ax \leq b$ is true, is its truth set and is also defined to be a closed half-space in R_n, the n-dimensional vector space.

Definition 2: Let $Ax \leq b$ be a closed half-space, the bounding hyperplane is the truth set of $Ax = b$.

Definition 3: A set C of vectors in R_n is said to be convex if, for every x and y in C, the vector $z = \alpha x + (1-\alpha)y$ also belongs to C, for all α, $0 \leq \alpha \leq 1$.

Definition 4: A polyhedral convex set is the intersection of a finite number of closed half-space, i.e., the truth set of simultaneous inequalities $Ax \leq b$.

Definition 5: Let a be a non-zero n-component column vector such that $a_i \neq 0$. Then P(a,i), the ith pivot matrix formed from a, is the matrix obtained by replacing the ith column of identity matrix I_{nxn} by the vector

$$\begin{bmatrix} -\frac{a_1}{a_i} \\ \vdots \\ -\frac{a_{i-1}}{a_i} \\ \frac{1}{a_i} \\ -\frac{a_{i+1}}{a_i} \\ \vdots \\ -\frac{a_n}{a_i} \end{bmatrix}$$

4.4.3 Pivoting Process

In solving simultaneous equalities, solutions can be obtained by repeatedly performing a pivoting operation on each column vector. In dealing with inequalities, the first step is to turn them into equalities.

Consider the inequality

$$Ax \leq b \tag{4.4.1}$$

together with the nonnegativity conditions

$$x \geq 0, \; b \geq 0. \tag{4.4.2}$$

We define an m-component column vector y of variable y_i, the slack variables, and consider:

$$Ax + y = b,\ x \geq 0 \text{ and } y \geq 0\ . \qquad (4.4.3)$$

Eq. (4.4.3) is equivalent to Eq. (4.4.1) together with Eq. (4.4.2), since if we have a solution to Eq. (4.4.1) and Eq. (4.4.2), then define

$$y = b - Ax \geq 0\ .$$

Conversely, a solution to Eq. (4.4.3) implies that $Ax \leq b$ since $y \geq 0$. A trivial solution is simply $x = 0$ and $y = b$.

In order to have a nontrivial solution such that at least some x_i's are positive, we will employ the process known as restricted pivoting.

Consider the following tableau for the systems of equations and inequalities of Eq. (4.4.3):

$$T^{(0)} = \begin{array}{|cccc|cccc|c|c|} \hline x_1 & x_2 & \cdots & x_n & y_1 & y_2 & \cdots & y_m & & \\ \hline a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 & b_1 & y_1 \\ a_{21} & a_{22} & & \cdot & 0 & 1 & & \cdot & b_2 & y_2 \\ \cdot & & & \cdot & \cdot & & & \cdot & \cdot & \cdot \\ \cdot & & & \cdot & \cdot & & & \cdot & \cdot & \cdot \\ \cdot & & & & \cdot & & & \cdot & \cdot & \cdot \\ a_{m1} & \cdot & \cdot & a_{mn} & 0 & \cdot & \cdot & 1 & b_m & y_m \\ \hline \end{array} \qquad (4.4.4)$$

We shall call $T^{(0)}$ the initial tableau and we have partitioned $T^{(0)}$ for easy viewing. To the right of $T^{(0)}$ are labels of basic variables. Initially, they are the slack variables. As the pivoting process progresses, some, maybe all, y_i's will be replaced by x_i's.

Let $T^{(1)}$ be constructed from $T^{(0)}$ by a pivoting process, and $T^{(j+1)}$ from $T^{(j)}$, etc. In constructing a next tableau, a new pivoting element and pivotal row must be chosen from the old tableau. The following restrictions must be observed:

R1: Do not choose a pivot in the last column.

R2: Let J be the index of the column to be performed a pivoting process on, i.e., "brought into the basis." For each i such that $t_{iJ} > 0$, compute b_i/t_{iJ}. Now, let $b_I/t_{IJ} = \text{Min}(b_i/t_{iJ})$, then row I is chosen to be the pivotal row and t_{IJ} is the pivoting element.

These two restrictions have the effect of preserving the nonnegativity of the last column throughout the pivoting process, hence the conditions $x \geq 0$, $y \geq 0$ are met.

4.4.4 Maximizing a Linear Function on a Convex Set

From previous discussions, the set of vectors x that satisfies $Ax \leq b$ forms a convex set. It is natural to inquire which particular vector from this set is the most desirable, namely, the optimal choice. In order to answer this question, we must state our criterion (objective), and that introduces an additional restriction over x. A simple objective function is a linear function of x. We may now formally state the problem as follows:

$$\begin{aligned} &\text{Maximize:} && cx \\ &\text{Subject to:} && Ax \leq b \\ & && x \geq 0 \\ & && b \geq 0 \end{aligned} \tag{4.4.5}$$

where c is a row vector. We can rewrite this problem as one involving only inequalities by replacing the maximizing statement with the inequality, $cx \geq z^0$, where z^0 is a parameter. The objective now is transformed into setting z^0 as large as possible such that the inequality system

$$\begin{aligned} Ax &\leq b \\ cx &\geq z^0 \\ x &\geq 0 \end{aligned} \tag{4.4.6}$$

has a solution. If we replace $cx \geq z^0$ by $-cx \leq -z^0$, then the initial tableau takes on the following form:

$T^{(0)} =$

x_1	x_2	$\cdots$	x_n	y_1	y_2	$\cdots$	y_m		
a_{11}	a_{12}	$\cdots$	a_{1n}	1	0	$\cdots$	0	b_1	y_1
a_{21}	a_{22}	$\cdots$	a_{2n}	0	1		0	b_2	y_2
.	.			.	.		.	.	.
.	.			.	.		.	.	.
.	.			.	.		.	.	.
a_{m1}	a_{m2}	$\cdots$	a_{mn}	0	0	$\cdots$	1	b_m	y_m
$-c_1$	$-c_2$	$\cdots$	$-c_n$	0	0	$\cdots$	0	0	cx

(4.4.7)

Notice that there are no slack variables introduced for the inequality $-cx \leq -z^0$. The reason is simply that since z^0 is a parameter, we can adjust it so that in the final solution, it would be satisfied as an equality. And we have set the initial value of z^0 to be 0. The last row of the tableau, except the last column, are called indicators. Initially, $x_1, x_2, \ldots, x_n$ are non-basic variables, and $y_1, y_2, \ldots, y_m$ are basic variables. At any stage of the pivoting process, the solution to (4.4.7) is obtained by setting all basic variables equal to the corresponding elements in the last column, and all nonbasic variables are set to zero. Thus, we have an initial solution of $x = 0$ and $y = b$, which is also a trivial one.

There are two additional restrictions we shall place in the pivoting process, namely:

R3: Do not choose a pivot element in the last row.

R4: Carry out the pivoting on the column only if its indicator is negative.

The effect of the last restriction is to make the pivoting process move from extreme point (of the convex set) to extreme point in such a way that the value of cx, i.e., the objective function, is monotonically increasing.

For visual clarity, we have partitioned the initial tableau (4.4.7) because of various restrictions placed on the last row and last column. But there is no other special connotation; we would still be referring to each individual element in the tableau as we normally would, with conventional matrix indexing. For example, $t_{m+1,\, m+n+1} = \sum_i c_i x_i$ at any stage of the pivot process with x_i the corresponding value at that stage.

Now we will say what a restricted pivoting process is. It is simply the premultiplication of tableau $T^{(j)}$ with a pivot matrix (see Definition 5, Section 4.4.2), formed after selecting a pivotal column and pivoting element from $T^{(j)}$ in conformance with restrictions R1 through R4.

4.4.5 A Recipe

The foregoing discussions can be fused into the following five steps with which the linear programming problem of (4.4.5) can be solved.

Step 1: Select a negative indicator; call this column J, the pivotal column.

Step 2: Calculate b_i/t_{iJ} for all i such that $t_{iJ} > 0$.

Choose the minimum value as b_I/t_{IJ}; the corresponding row I is the pivotal row, and t_{IJ} is the pivoting element.

Step 3: Form a new tableau by carrying out the pivoting process.

Step 4: Replace y_I, corresponding to pivotal row I, by x_J, corresponding to pivotal column J. Now, x_J becomes a basic variable and the particular step is sometimes being referred to as "bringing x_J into the basis", while y_I is "going out of the basis".

Step 5: Repeat steps 1 through 4 until all indicators are nonnegative. Set all the nonbasic variables to zero. The rest of the answer, the basic variables, are the elements of $t_{i,m+n+1}$, for $i = 1,2,\ldots,m$. $t_{m+1,\ m+n+1} = \sum_i c_i x_i$ and is the maximum over the truth set of (4.4.5).

The algorithm so described is known as the Simplex Method. Note that, in Step 2, if there is no $t_{iJ} > 0$, $1 \leq i \leq m$, it means that the maximum problem has an unbounded solution.

4.4.6 Direct Problem and Its Dual

A linear programming problem requires the maximization or minimization of a linear function of variables subject to linear inequality constraints. In specific applications, the original problem of interest may be either a maximizing or a minimizing problem from which the data matrices, A, b, and c may be derived. If we formulate one as Direct (Primal) Problem, be that as a maximizing or minimizing, then the other would be its Dual. We have been using the maximizing problem to carry the discussion, so we will designate this to be our Direct Problem, without loss of generality. Their respective canonical forms can be stated as the following:

Direct Problem

$$\begin{aligned} &\text{Maximize:} && cx \\ &\text{Subject to:} && Ax \leq b \\ & && x \geq 0 \end{aligned} \qquad (4.4.8)$$

Dual Problem

$$\begin{aligned} &\text{Minimize:} && wb \\ &\text{Subject to:} && wA \geq c \\ & && w \geq 0 \end{aligned} \qquad (4.4.9)$$

Note that w is a row vector, the dual to the original variable x, a column vector. Clearly, there are slack variables z (a $1 \times n$ vector of variables z_i) which are dual to slack variables y (an $m \times 1$ vector of variables y_i). The equality forms of both problems are the following:

$$\begin{aligned} &\text{Maximize:} && cx \\ &\text{Subject to:} && Ax + y = b \\ & && x \geq 0,\ y \geq 0 \end{aligned} \tag{4.4.10}$$

$$\begin{aligned} &\text{Minimize:} && wb \\ &\text{Subject to:} && wA - z = c \\ & && w \geq 0,\ z \geq 0 \end{aligned} \tag{4.4.11}$$

And we will label the initial tableau with all the variables as follows:

$$T(0) = \begin{array}{c|cccc|cccc|c|c} & x_1 & x_2 & \cdots & x_n & y_1 & y_2 & \cdots & y_m & & \\ \hline & a_{11} & a_{12} & \cdots & a_{1n} & 1 & 0 & \cdots & 0 & b_1 & y_1 \\ & a_{21} & a_{22} & & & 0 & 1 & & 0 & b_2 & y_2 \\ & \vdots & \vdots & & & \vdots & \vdots & & \vdots & \vdots & \vdots \\ & a_{m1} & a_{m2} & \cdots & a_{mn} & 0 & 0 & \cdots & 1 & b_m & y_m \\ \hline & -c_1 & -c_2 & & -c_n & 0 & 0 & & 0 & 0 & cx \\ \hline & z_1 & z_2 & \cdots & z_n & w_1 & w_2 & \cdots & w_m & wb & \end{array} \tag{4.4.12}$$

In this tableau, we see that both cx and wb point to the same entry in the tableau. This suggests that the objective functions of both problems are equal, namely, zero, at the beginning, and perhaps also at some other points. In fact, they are also equal at the optimum point. The following important theorem states this fact.

Duality Theorem: The maximum problem has as a finite solution a vector x^0 such that $cx^0 = \text{Max } (cx)$ if and only if the minimum problem has a finite solution a vector w^0 such that $w^0 = \min (wb)$. Moreover, $cx^0 = w^0 b$.

This theorem was first conjectured by John von Neumann and later was stated and proved by Gale, Kuhn, and Tucker [36]. We will utilize the result of this theorem in the next chapter. Furthermore, Simplex Method solves both problems at once.

CHAPTER V

Resource Allocation under Linear Constraints

5.1 Introduction

In Chapter 4 we have singled out, among all the system activities, the resource allocation activity as the focus of our attention. Resources come in many types. In this chapter we will examine a specific type of resource, namely main memory. There are two basic "commodities" a program must acquire before any computation can proceed. They are memory space and a processing unit. In the current discussion, we will not consider the processor requirement as critical as the memory space in the sense that a processor can be switched from job to job with relative ease, albeit at the expense of a certain amount of overhead, a relatively small amount at that. Memory then has become the most important and most scarce of all resources. In a simplified viewpoint, we equate the allocation of memory space to a program to that of initiating that particular program from system job queue to system ready queue. Using a common, long-established terminology, we would say that this is "loading" a program into the memory. Section 2 discusses the criterion with which programs are selected to be loaded. If we consider that the loading action happens at discrete points in time and that at each occurrence of this action, memory will be filled to the extent possible, according to some goal, then this action is static in nature. That is, the goal is either satisfied or not, at the moment of loading, and not over a period of time. If we state our criterion in the form of a linear objective function, the Simplex Method provides the answer. However, practicality prevails in the consideration of program loading. The inherent difficulty in obtaining the optimal solution is briefly discussed, and an heuristic approach is therefore suggested. In an effort to present a more general view on the optimization criterion for our goal oriented allocation activity, we also discuss the concept of value. In Section 3, we focus our attention at the question of memory utilization over a period of time. This is similar to the question of space utilization in a warehouse management problem, and the behavior is dynamic in nature. Various ramifications will also be discussed.

5.2 Program Loading - Static Planning

5.2.1 General Formulation

Although a program may require many different types of resources before

the execution can be commenced, none will be more critical than memory space. The assumption is, of course, that every program will receive its share of processor time. Therefore, the main concern in resource allocation problem is memory space. In other words, we choose to look at the system allocation activity as primarily, at least for the moment, as activity which distributes primary commodities (memory spaces) among the basic activities (individual user programs) to achieve productions (computations). The purpose is then to devise a way (a plan) to allocate those available memory spaces such that the computer system may execute programs according to some policy (goal oriented).

Let x_i represent an individual program or one of its steps, and $(x_1,x_2,\ldots,x_n)$ the collection of such programs or program steps. If we follow the idea as set forth in Chapter 4 and look upon x_i as a basic activity, and if associated with x_i there is a number λ_i, the "level" of the activity, then the total activity, i.e., the allocation activity, is constrained by the total resources, M. It is

$$\lambda_1 x_1 + \lambda_2 x_2 + \ldots + \lambda_n x_n \leq M \ . \tag{5.2.1}$$

The interpretations of the variables are such that $\lambda_i x_i$ represents the individual memory requirement of program x_i and M is the total memory space available. We may consider λ_i to be the size (maximum memory units required per basic activity) of program x_i. Then the following must be true:

$$0 \leq x_i \leq 1 \ . \tag{5.2.2}$$

This means that x_i assumes the value between 0 and 1. We further state that the goal of our allocation activity is to plan our use of the memory spaces that a certain linear function, namely, the objective function, is maximized. This objective function has the general form of

$$c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \ . \tag{5.2.3}$$

Where c_i's are constants. The canonical form of this maximization problem is the following:

$$\begin{aligned} &\text{Maximize:} && cx \\ &\text{Subject to:} && Ax \leq b \\ & && 0 \leq x_i \leq 1, \quad x_i \text{ integer} \end{aligned} \tag{5.2.4}$$

where

$$A = \begin{bmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_n \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}, \quad b = \begin{bmatrix} M \\ 1 \\ \vdots \\ 1 \end{bmatrix}, \quad c = \text{a constant row vector,}$$

We can devise a specific optimal allocation plan only if the objective function, i.e., the row vector c, is explicitly defined.

5.2.2 Illustration

Let us consider the set of programs x_1, x_2, x_3 with memory requirements 40, 30, and 20 respectively, with total memory equal to 60 units, i.e.

Program	Size	Total Memory
x_1	40	M = 60
x_2	30	
x_3	20	

Note that the program size and the total memory are of the same unit, such as words, blocks, or pages. Following Eq. (5.2.1), we can write down:

$$40x_1 + 30x_2 + 20x_3 \leq 60 \ . \tag{5.2.5}$$

In addition to the resource constraints, we also have the condition stated in Eq. (5.2.2). Combining all the constraints, we may write down a set of simultaneous inequalities as follows:

$$\begin{aligned} 40x_1 + 30x_2 + 20x_3 &\leq 60 \\ x_1 &\leq 1 \\ x_2 &\leq 1 \\ x_3 &\leq 1 \end{aligned} \tag{5.2.6}$$

Clearly, the structural matrix A and the constrain vector b assume the following values:

$$A = \begin{bmatrix} 40 & 30 & 20 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 60 \\ 1 \\ 1 \\ 1 \end{bmatrix} \tag{5.2.7}$$

What remains to be specified is our objective function.

A. Case I

If our objective for the memory allocation is to pack as many programs as possible, i.e., maximum degree of multi-programming, then we may write down the objective function as:

$$\text{Maximize: } x_1 + x_2 + x_3 \tag{5.2.8}$$

The row vector c thus becomes:

$$c = [1, 1, 1] \tag{5.2.9}$$

Combining (5.2.7) and (5.2.9), adding proper slack variables, we may construct the initial tablues $T^{(0)}$ in the form of (4.4.12) as follows:

$T^{(0)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
40	30	20	1	0	0	0	60	y_1
1	0	0	0	1	0	0	1	y_2
0	1	0	0	0	1	0	1	y_3
0	0	1	0	0	0	1	1	y_4
-1	-1	-1	0	0	0	0	0	

In this first example, we will present the recipe as stated in Chapter 4 in sufficient details, not only to illustrate the Simplex Method but also by studying the step-by-step procedures, we can see clearly <u>how</u> an optimal plan may be achieved. The algorithm as outlined in Chapter 4 treats the variables as continuous ones. We will treat the variables as continuous in this example. The changes that must be made to the solution vectors if we are interested in integer values will also be discussed.

(1) We have three negative indicators, namely, t_{51}, t_{52}, t_{53}. Suppose we choose t_{53}. Then column 3 is the pivotal column.

(2) Compute $\frac{b_i}{t_{i3}}$ for all i such that $t_{i3} > 0$. They are

$$\frac{b_1}{t_{13}} = \frac{60}{20} = 3; \quad \frac{b_4}{t_{43}} = 1. \quad \text{Min } (\frac{b_i}{t_{i3}}) = \frac{b_4}{t_{43}} = 1.$$

Therefore, t_{43} is the pivotal element and Row 4 is the pivotal row.

(3) Form a new tableau, $T^{(1)}$, by carrying out the pivoting process. Pivotal element and pivotal row are indicated by circle and arrow respectively.

(4) Since Column 3 and Row 4 are pivotal, x_3 replaces y_4 to the right of $T^{(1)}$. That is, x_1 has been brought into the basis and become a basic variable while y_4 has been displaced and gone out of the basis. Therefore it has become a non-basic variable.

The above four steps correspond to the recipe given in Chapter 4, Section 4.4.5. The resulting tableau, after one pivoting process, becomes:

$T^{(1)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
40	30	0	1	0	0	-20	40	y_1
1	0	0	0	1	0	0	1	y_2
0	1	0	0	0	1	0	1	y_3
0	0	(1)	0	0	0	1	1	x_3 ←
-1	-1	0	0	0	0	1	1	cx

From this tableau, we see that a "feasible" solution is the following:

$$x = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \quad y = \begin{bmatrix} 40 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

And the objective function at this stage is:

$$\sum_i c_i x_i = 1 \cdot 0 + 1 \cdot 0 + 1 \cdot 1$$
$$= 1$$

Since there are two more negative indicators left, namely t_{51} and t_{52}, we have to repeat the above four steps. Suppose we choose t_{52}. Then Column 2 is the pivotal column and we have chosen t_{32} as the pivoting element and Row 3, the pivotal row. Again, we obtain a new tableau, $T^{(2)}$ in the following:

$T^{(2)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
40	0	0	1	0	-30	-20	10	y_1
1	0	0	0	1	0	0	1	y_2
0	(1)	0	0	0	1	0	1	x_2 ←
0	0	1	0	0	0	1	1	x_3
-1	0	0	0	0	1	1	2	cx

The feasible solution at this point is:

$$x = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \quad y = \begin{bmatrix} 10 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

And the objective function has increased to:

$$\sum_i c_i x_i = 1 \cdot 0 + 1 \cdot 1 + 1 \cdot 1$$
$$= 2$$

Since there is one more negative indicator left, i.e., t_{51}, there is one more pivoting process to be carried out, according to the algorithm. We have t_{11} as pivoting element, Row 1 as pivotal row. Next tableau, $T^{(3)}$, is obtained:

$T^{(3)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
①	0	0	$\frac{1}{40}$	0	$\frac{3}{4}$	$-\frac{1}{2}$	$\frac{1}{4}$	x_1 ←
0	0	0	$-\frac{1}{40}$	1	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	y_2
0	1	0	0	0	1	0	1	x_2
0	0	1	0	0	0	1	1	x_3
0	0	0	$\frac{1}{40}$	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{9}{4}$	cx

Since there are no more negative indicators left, $T^{(3)}$ is the final tableau. The feasible solution, which is also the optimal solution, is now:

$$x = \begin{bmatrix} \frac{1}{4} \\ 1 \\ 1 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ \frac{3}{4} \\ 0 \\ 0 \end{bmatrix}$$

and the objective function $\sum_i c_i x_i = \frac{9}{4}$ is the maximum possible under the constrains as stated in (5.2.6).

There are various ways to deal with the Integer Programming problem if we desire an integer solution. If the number of variables are not too large, then one of the most effective is the Branch-and-Bound method [37]. This method

is essentially an implicit enumeration on all the feasible solutions and chooses the solution that optimizes the objective function. Without going into the details on Integer Programming, it suffices to note that, for this particular example, the pivoting process stops after reaching tableau $T^{(2)}$, with a feasible, which is also optimal, solution of:

$$x = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \quad y = \begin{bmatrix} 10 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

and the objective function has a value of 2.

Recall that y_i's are slack variables. In particular, y_1 is complementary to the resources requirement, and y_2 is complimentary to x_1. Here we have $y_1 = 10$; $x_1 = 0$, $y_2 = 1$; $x_2 = 1$, $y_3 = 0$; $x_3 = 1$, and $y_4 = 0$. This result indicates that the maximum degree of multi-programming achieved is 2; programs to be loaded are x_2 and x_3, with x_1 idle; 10 units of memory remain unused.

We have not discussed the "sequence" with which the negative indicators were chosen. What we have done is to have brought the x_i's into basis in the order of x_3, x_2, and x_1. Suppose we now choose the order x_1, x_3, and x_2. The subsequent results are somewhat different. They are:

$T^{(1)}$ =

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
0	30	20	1	-40	0	0	20	y_1
(1)	0	0	0	1	0	0	1	x_1 ←
0	1	0	0	0	1	0	1	y_3
0	0	1	0	0	0	1	1	y_4
0	-1	-1	0	1	0	0	1	cx

$T^{(2)}$ =

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
0	30	0	1	-40	0	-20	0	y_1
1	0	0	0	1	0	0	1	x_1
0	1	0	0	0	1	0	1	y_3
0	0	(1)	0	0	0	1	1	x_3 ←
0	-1	0	0	1	0	1	2	cx

$T^{(3)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
0	①	0	$\frac{1}{30}$	$-\frac{4}{3}$	0	$-\frac{2}{3}$	0	$x_2 \leftarrow$
1	0	0	0	1	0	0	1	x_1
0	0	0	$-\frac{1}{30}$	$\frac{4}{3}$	1	$\frac{2}{3}$	1	y_3
0	0	1	0	0	0	1	1	x_3
0	0	0	$\frac{1}{30}$	$-\frac{1}{3}$	0	$\frac{1}{3}$	2	cx

$T^{(4)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
0	1	0	0	0	1	0	1	x_2
1	0	0	$\frac{1}{40}$	0	$-\frac{3}{4}$	$-\frac{1}{2}$	$\frac{1}{4}$	x_1
0	0	0	$-\frac{1}{40}$	①	$\frac{3}{4}$	$\frac{1}{2}$	$\frac{3}{4}$	$y_2 \leftarrow$
0	0	1	0	0	0	1	1	x_3
0	0	0	$\frac{1}{40}$	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{9}{4}$	cx

In their final forms, the optimal solutions are identical, regardless of what sequences we choose to bring the variables into the basis as it should be, if we treat this as a continuous variable problem. However, it takes four steps to reach the final solution while only three were needed in the first instance. As pointed out in Chapter 4, the restricted pivoting process solves the maximization problem by moving from one extreme point to another, within the polyhedral convex set, such that the linear objective function is monotonically increasing. A different sequence in bringing the variables into the basis simply means that we move in a different route, along the bounding hyperplane, toward the optimal solution.

Again, if we restrict ourselves to integer valued solutions, the process should terminate at tableau $T^{(2)}$ with a solution of:

$$x = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

Although the optimized value is the same, namely, the maximum degree of

multi-programming is 2, but the choice of the programs is different, namely x_1 and x_3. Also, the two solutions do differ in the amount of memory they used.

B. Case II

Suppose, associated with each individual program, there is a "value". Specifically:

Program	Size	Value
x_1	40	70
x_2	30	50
x_3	20	30

Each value shown here is a quantification of the relative importance of each program. In this light, case I can be considered as a special case in that all programs are of equal importance.

Let us state that the goal is to find a subset of programs to load into the memory such that the total values are at a maximum. The objective function becomes:

$$\text{Maximize: } 70x_1 + 50x_2 + 30x_3 \tag{5.2.10}$$

with a row vector c of:

$$c = [70, 50, 30] \tag{5.2.11}$$

and the initial tableau is the following:

$T^{(0)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
40	30	20	1	0	0	0	60	y_1
1	0	0	0	1	0	0	1	y_2
0	1	0	0	0	1	0	1	y_3
0	0	1	0	0	0	1	1	y_4
-70	-50	-30	0	0	0	0	0	cx

If we choose t_{51} as the first indicator, then Column 1 is pivotal: t_{21} is the pivotal element; and Row 2 is the pivotal row. Carrying out the pivoting process, we obtain tableau $T^{(1)}$.

$T^{(1)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
0	30	20	1	-40	0	0	20	y_1
①	0	0	0	1	0	0	1	x_1 ←
0	1	0	0	0	1	0	1	y_3
0	0	1	0	0	0	1	1	y_4
0	-50	-30	0	70	0	0	70	cx

If we treat this as a continuous variable problem, we may choose Column 2 as pivotal and obtain tableau $T^{(2)}$:

$T^{(2)} =$

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
0	①	$\frac{2}{3}$	$\frac{1}{30}$	$-\frac{4}{3}$	0	0	$\frac{2}{3}$	x_2 ←
1	0	0	0	1	0	0	1	x_1
0	0	$-\frac{2}{3}$	$-\frac{1}{30}$	$\frac{4}{3}$	1	0	$\frac{1}{3}$	y_3
0	0	1	0	0	0	1	1	y_4
0	0	$\frac{10}{3}$	$\frac{5}{3}$	$\frac{10}{3}$	0	0	$\frac{310}{3}$	cx

Since there is no negative indicator left, the pivoting process terminates and the feasible solution is:

$$x = \begin{bmatrix} 1 \\ \frac{2}{3} \\ 0 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{3} \\ 1 \end{bmatrix}$$

which is also optimum, with a total value of:

$$\begin{aligned} \sum_i c_i x_i &= 70x_1 + 50x_2 + 30x_3 \\ &= 70 \cdot 1 + 50 \cdot (\tfrac{2}{3}) + 30 \cdot 0 \\ &= 70 + \frac{100}{3} \\ &= \frac{310}{3} \end{aligned}$$

However, if we desire the integer solution, then we should choose Column 3 from $T^{(1)}$ as pivotal column instead, and obtaining $T^{(2)}$ as shown in the following:

x_1	x_2	x_3	y_1	y_2	y_3	y_4	b	
0	$\frac{3}{2}$	(1)	$\frac{1}{20}$	-2	0	0	1	x_3 ←
1	0	0	0	1	0	0	1	x_1
0	1	0	0	0	1	0	1	y_3
0	$-\frac{3}{2}$	0	$-\frac{1}{20}$	2	0	1	0	y_4
0	-5	0	$\frac{3}{2}$	10	0	0	100	cx

Therefore, in the integer case, we have as following the optimal solution:

$$x = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad , \quad y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

and the objective function has the value of:

$$\begin{aligned} \sum_i c_i x_i &= 70x_1 + 50x_2 + 30x_3 \\ &= 70 \quad 1 + 50 \quad 0 + 30 \quad 1 \\ &= 100 \end{aligned}$$

5.2.3 Interpretations and Related Questions

We have tabulated the results of this illustrative example under different objective functions in Table 5.1. By restricting our attention integer valued solutions, we are, in effect, considering allocation in a non-paging environment. The solution vector consists of either 0's or 1's, signifying the absence or presence of certain programs in the optimal allocation plan. In Case I, there are two possible solutions. Each can be considered as optimal solution since each achieves degree 2, which is the maximum possible under the constraints. We have arrived at these two solutions by different pivoting sequences. Furthermore, if the final solution represents a "plan" to achieve an optimal goal, then the existence of more than one final solution means that we have alternate paths which lead to the optimum. This illustrates the point made in Chapter 4, when we stated that in mathematical programming, objectives must be stated in terms of basic ends, thus permitting the consideration of alternative means. In this case, our specific objective is to achieve the maximum degree of multi-programming. We can achieve this by

	Optimal Solution			Objective		
	x_1	x_2	x_3	Degree of Multiprogramming	Value	Unused Memory Units
Case I	0	1	1	2		10
	1	0	1	2		0
Case II	1	0	1		100	0

Table 5.1 Results of the Illustrative Example

either loading programs x_2 and x_3, or alternatively, by loading x_1 and x_3. The geometric details are shown in Figure 5.1. One of the solution vectors lies on the bounding hyperplane of the memory space constraint while the other does not. This is why in the former case there are no unused memory units while in the latter, there are 10 units of space left. It would seem that the solution which utilizes all memory spaces is a better choice although one cannot make such a judgement a priori. On the other hand, a very practical consideration would favor one solution over the other simply because this particular solution, or the optimal allocation plan, can be easily devised. This is not to say that given two optimal solutions, such as the ones listed in Table 5.1, one is easier to implement than the other; the difference lies in how the solutions were obtained. However, it is highly impractical for the operating system to set up and solve maximizing problems in the form of (5.2.4) everytime the system has to carry out the allocation activity. The point we want to make clear is that to arrive at an optimum policy, i.e., finding the solution which solves the programming problem of (5.2.4), is not necessarily the same as implementing it. Naturally, it is very desirable to have a simple allocation algorithm which, when carried out, would constitute an optimum strategy; a strategy that stems from a predefined objective.

In light of this discussion and if we look at the processes by which the two solutions of Case I were obtained, we may state that the first solution can be obtained by selecting the smallest program first, namely, the allocation unit x_3, and then x_2. Since x_1 no longer can fit into the remaining space, we stop. The second solution can be arrived at by selecting the largest program first, namely, the unit x_1. Since x_2 no longer can fit into the remaining space, we by pass it and select x_3. However, one implicit restriction on the loading of programs is that we cannot load and unload programs in order to find the best combination such that a maximum number of them can be loaded; something that cannot be explicitly formulated in the linear programming problem. For example, if we change the size of x_1 to 50 units instead of 40, then the procedure associated with the first solution, namely, smallest first, would still attain a multi-programming of degree 2, while the procedure for the second solution, namely, largest program first, could only select x_1 and no more; degree of multi-programming is therefore degraded to 1. The reason is simply that after loading x_1, the remaining space (10 units) is not enough to load either x_2 or x_3 and we cannot unload x_1 to try for other

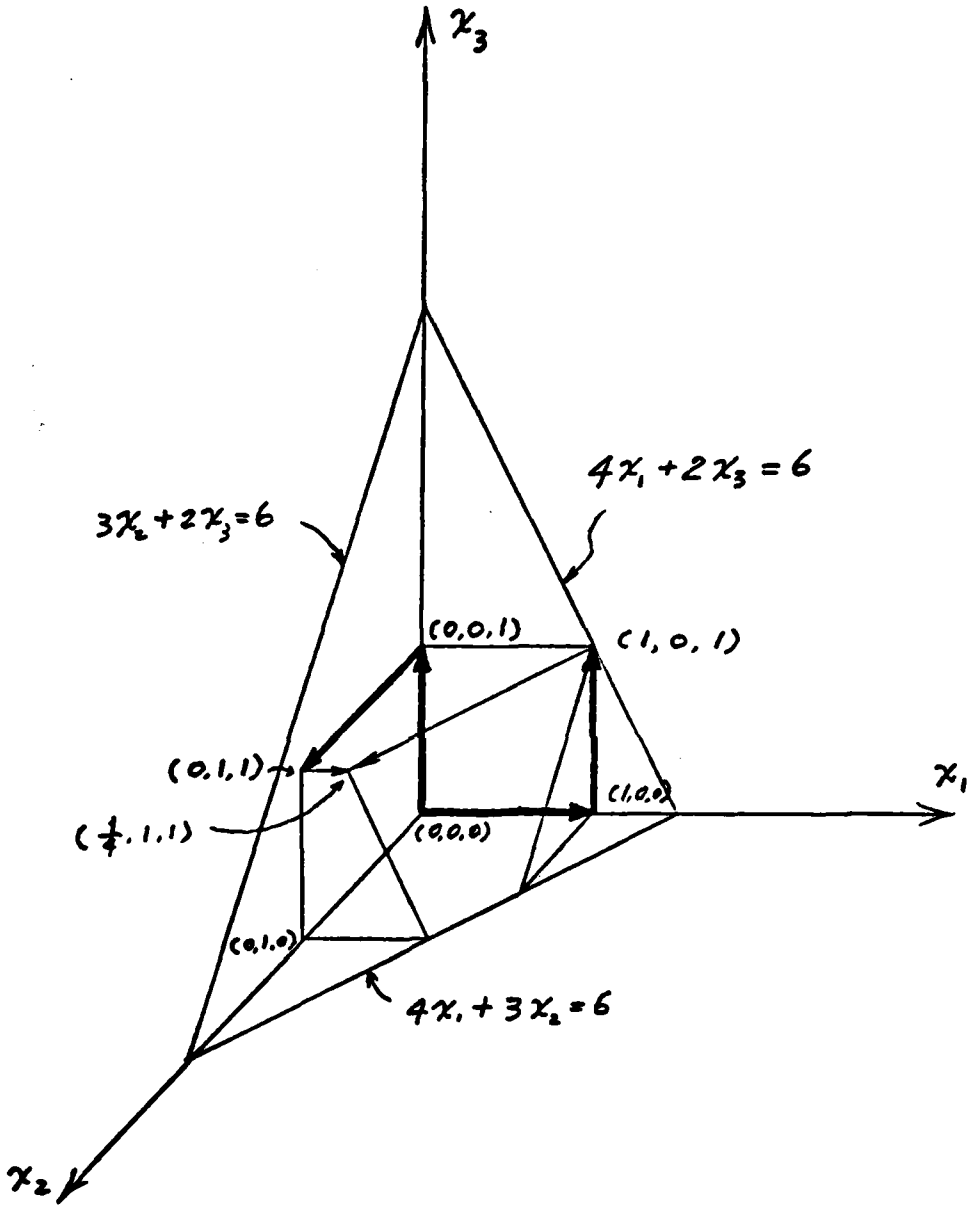


Fig. 5.1 Geometric interpretation of the simplex calculation

selections, due to practical considerations. Hence, we may state the following:

Proposition 5.2.1: If λ_i is the size of program x_i, and if $\lambda_1 < \lambda_2 < \ldots < \lambda_j < \lambda_{j+1} < \ldots < \lambda_n$, then maximum degree of multiprogramming can be achieved for a given memory size by loading first x_1 then x_2 in ascending order until the remaining space cannot fit the next larger program.

Proposition 5.2.2: If first j programs are loaded into the memory, and if (j + 1)th program cannot fit in, then the unused memory space is at most $(\lambda_{j+1} - 1)$ units.

One can see from Proposition 5.2.2 that if one follows the algirithm outlined in Proposition 5.2.1, the unused memory space is potentially large. To reduce this potentially large amount of unused space, it is intuitively clear that one should deal with a large pool of small programs. Indirectly, this supports the argument of a paged machine, although the decision for a paged machine is a result of many other considerations [8].

In Case II, we have attached a "value" to each of the individual programs, without elaborating on how each value was arrived at. We will give a formal discussion in Section 5.2.5. The Simplex Method would give a satisfactory solution if treated as a continuous variable case. But, a straightforward procedure by which an optimal solution can always be achieved, i.e., a feasible solution that maximizes the total value, is what actually is desired. In this case, it is more involved. To illustrate the problem area, let us try the following approach: load the program with the largest "value" first, which, in this case, is x_1, and then try x_2. Since x_2 cannot fit into the remaining space which is only 20 units, we end up loading x_3, thus obtaining an optimal solution as listed in Table 5.1. However, if we change the size of x_1 to 50 units without changing the associated value, namely 70, the procedure just described would fail to maximize the total value of the programs loaded for a given memory space. The relation between the program size and its associated value must be taken into consideration.

Suppose we follow the largest-ratio-first rule, then from Table 5.2, we see that x_2 will be the first candidate considered for loading, and then x_3. If we only consider integer solutions, loading x_2 and x_3 with a total value of 80 would be the best possible choice. Furthermore, if we allow a fraction of a program to be loaded, i.e., x_i's are treated as continuous variables, then the largest-ratio-first rule will always give the maximum total value.

This is clear from the Simplex Method; loading a program means, in the context of restricted pivoting process, selecting that particular indicator to be "brought into the basis", as discussed in Chapter 4. However, this is, in general, not true for integer solutions. Consider Table 5.3.

According to largest-ratio-first rules, we would load x_1 as first choice. If the total memory space is only 60 units, there are only 19 units left, and not enough to load either x_2 or x_3. Since reality dictates that one does not unload x_1 after loading it into the memory, we are stuck with the result of loading only one program with an objective function valued at 79, while, in fact, we could do better by loading x_2 and x_3 instead, with a total of 80.

Branch-and-Bound method can, of course, handle this situation. However, this method essentially involves implicit enumeration and when the number of candidates on the system job queue (see Figure 4.2.1) is large, it becomes cumbersome. It touches upon the question of how one might classify the degree of difficulties of a given problem. This will be taken up in the next section.

5.2.4 Inherent Difficulties and Heuristic Approach

From our discussion in previous sections, we see that solving the linear programming problem has been reduced by practical considerations, to a sequencing problem. That is, by what sequence should one choose the negative indicators to be brought into the basis so that when the process terminates, the objective function is at a maximum? The analogy is clear if we think of the "sequence of negative indicators" as the "sequence of programs" to be loaded into the memory.

We are dealing with finite number of programs. Therefore, the number of choices for program loading sequence is finite. Obviously, there is a finite algorithm for finding a sequence that is optimal in some sense; one can always perform an exhaustive search over the finite elements in question. But the degree of difficulty, or cost, in applying an algorithm makes it significant the question of the existence of a better-than-finite algorithm. The relative cost, in time or whatever, is a fairly clear notion. The problem-domain of applicability for an algorithm often suggests for itself possible measure of size for the individual problems. Once a measure of problem size is chosen, one can define the order of difficulty of algorithm as the least upper bound on the cost of applying such an algorithm [49].

If the relative cost, in time or number of steps, of applying algorithm

Programs	Size	Value	Value/Size
x_1	50	70	1.40
x_2	30	50	1.67
x_3	20	30	1.50

Table 5.2 Example

Programs	Size	Value	Value/Size
x_1	41	79	1.92
x_2	30	50	1.67
x_3	20	30	1.50

Table 5.3

A to a problem of size N is bounded by

$$aN^b$$

where a, b are constants, then we say that the problem is polynomial bound. Otherwise, it is polynomial complete. In other words, a polynomial bound problem has an algorithm that is terminated in polynomial bounded time (or steps). Such an algorithm is called fast or simple.

There exists a large class of combinatorial problems belonging to the class of polynomial complete problems. Knapsack packing is one of them. It has also been conjectured that no polynomial complete problem has a fast algorithm. We have used "fast" in the same sense as "simple", and it means that the algorithm terminates in a polynomial bounded time. Clearly, the memory allocation problem (or program loading problem, as we have phrased it) is similar to the Knapsack problem if we view each individual program as an indivisible object, with program "size" being the "weight", and the total memory space being the weight capacity that a person is able to carry. A given memory space cannot accommodate all the programs competing for it, just as a person cannot carry all the accumulated weight in the knapsack. The analogy is complete when a value is attached to each program, just as the case of knapsack packing when each object is being given a value, a value that is meaningful to the person who carried the knapsack. Knowing that the knapsack problem is a polynomial complete problem should convince us that a simple algorithm to find the loading sequence of programs that would maximize a given objective function, does not exist. Under the circumstances, we should appeal to our intuition based on our understanding of the complexity of the problem. Namely, we resort to heuristics.

Let us assume that we have a pool of programs ($x_1 \ldots x_n$) which are to be considered for memory space allocation. Associated with each x_i, there is a value of c_i, and the value-to-size ratio can be formed. Let us further suppose that this community of programs are put into a sorted list according to the magnitude of each value-to-size ratio in descending order. This sorted list has n items with the top and the bottom each corresponding to the largest and the smallest ratio, respectively. The relative positions of, or the index to, this sorted list signifies the magnitude of each value-to-size ratio relative to each other. For $n \leq 2$, the sequencing problem is obvious. For $n \geq 3$, the algorithm is shown in the form of a flow chart in Figure 5.2.

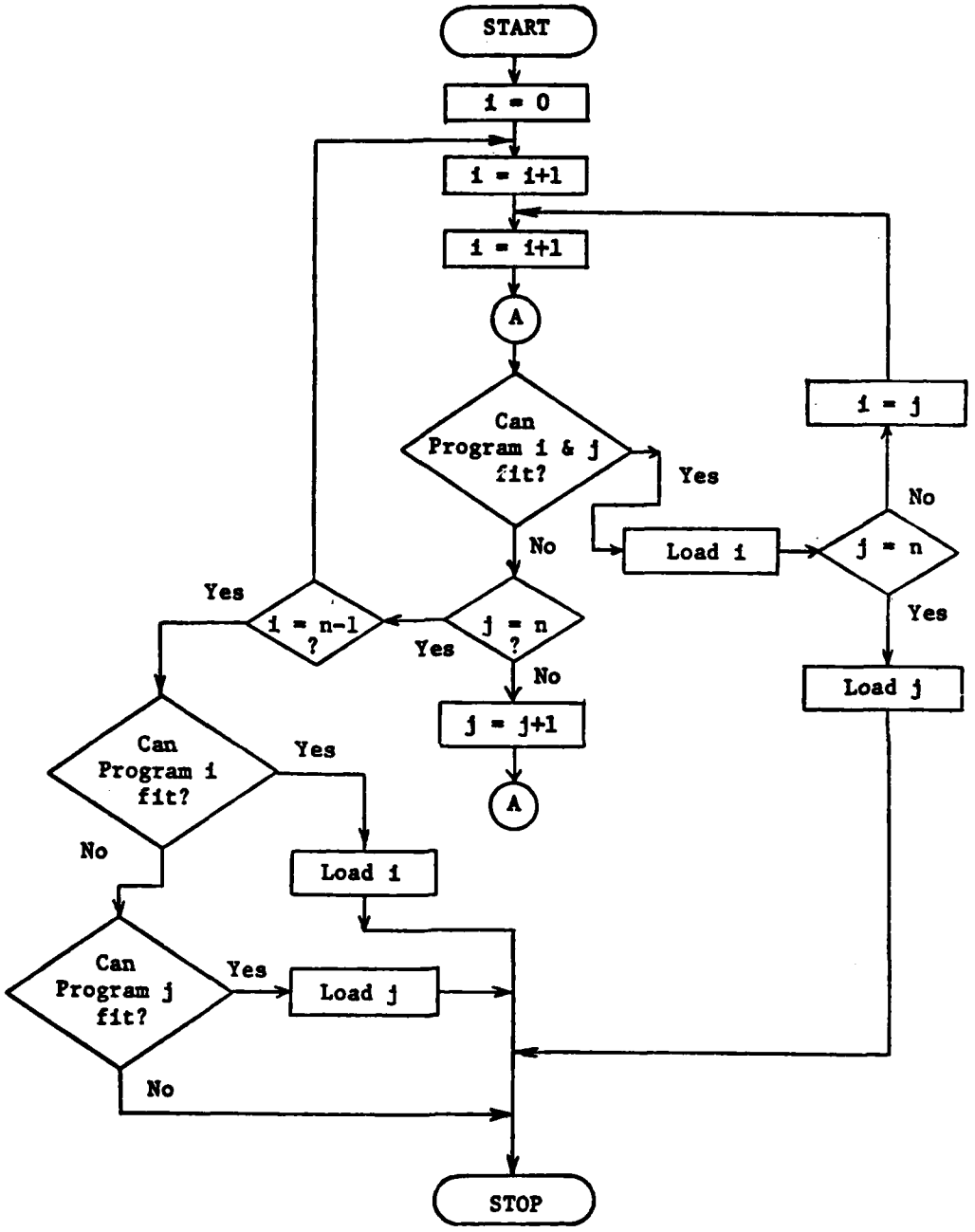


Fig. 5.2 Flow chart of a heuristic algorithm for finding program loading sequence, for $n \geq 3$.

i) at least one program can fit in.

ii) $\sum_{i=1}^{n} \lambda_i x_i \geq M$, i.e., total memory requirement of all the programs waiting for allocation is at least the amount of total available memory space.

iii) We are working with a sorted list, i.e.:

$$(\frac{c_k}{\lambda_k})_i > (\frac{c_p}{\lambda_p})_j \quad \text{if } i > j$$

where i, j are indices (positions) on this list.

iv) In the flow chart, "load j" means to load the program that is in the jth position on the list.

Let us consider yet another example as shown in Table 5.4. We reorganize Table 5.4 into a sorted list in descending order of the v/s ratio, as shown in Table 5.5. Assuming that these six programs are on the system job queue waiting to be assigned memory space, and if the available memory space M = 65 units, then our goal is to load these programs that would fit into the memory with as large a sum of total value as possible. Note that:

i) $M > \lambda_i x_i \quad i = 1, 2, 3, 4, 5, 6$

i.e., at least one program can fit in.

ii) $\sum_{i=1}^{6} \lambda_i x_i = 30 + 27 + 25 + 20 + 18 + 15$
$= 135$
$> M ; M = 65$

iii) $(\frac{c_k}{\lambda_k})_i > (\frac{c_p}{\lambda_p})_j \quad \text{if } i > j$

i.e., for $i = 2 \quad c_k = c_4 = 61 \quad \lambda_k = \lambda_4 = 27$

$$(\frac{c_4}{\lambda_4})_2 = \frac{61}{27} = 2.26$$

$j = 4 \quad c_p = c_1 = 40 \quad \lambda_p = \lambda_1 = 20$

$$(\frac{c_1}{\lambda_1})_4 = \frac{40}{20} = 2.00$$

Program	Size	Value	Value/Size
x_1	20	40	2.00
x_2	25	54	2.16
x_3	18	31	1.72
x_4	27	61	2.26
x_5	30	70	2.33
x_6	15	25	1.66

Table 5.4 Example

Index	Program	Size	Value	Value/Size
1	x_5	30	70	2.33
2	x_4	27	61	2.26
3	x_2	25	54	2.16
4	x_1	20	40	2.00
5	x_3	18	31	1.72
6	x_6	15	25	1.66

Table 5.5 Sorted According to V/S Ratio

Program	Size	Value	Value/Size
x_1	41	81	1.97
x_2	30	50	1.67
x_3	20	30	1.50

M = 60 units

Table 5.6 Example

It can be verified that if we apply the algorithm as in the flow chart in Figure 5.2, we would end up loading program indices 1, 4, 6 which correspond to programs x_5, x_1, and x_6. In this case, this choice also happens to be optimal, i.e., that total value (70 + 40 + 25) = 135 is the largest possible under the memory constraint. However, consider Table 5.6.

It can be shown that the algorithm in Figure 5.2 would select x_2 and x_3 to be loaded with a total value of 80, while the optimum choice would have been x_1 with a value of 81. Clearly, the algorithm as proposed is a sub-optimal one. We have pointed out earlier that the complexity of problems of this nature is polynomial complete. In the pursuit of a simple program loading algorithm, i.e., the algorithm itself terminates in polynomial bounded time, we have sacrificed the possibility of obtaining the optimal feasible solution to our linear programming problems at all times.

5.2.5 The Value Concept

In our previous discussions, we have used the term "value" freely, without actually elaborating on it. Also, we have seen that under similar circumstances, the formulation of different objective functions could lead to different allocation plans. In Chapter 3, we have sought to broaden our view on the question of performance. In Chapter 4, we have further propounded the notion of goal oriented allocation. What is this goal? It is clear from the context that we have chosen our goal to be the maximization of a given set of possible values. The programs thus selected (allocated) would be of the utmost valuation to the system if actually processed. Therefore, the whole question of system performance is tied to the resource allocation problem through the determiniation of a general objective function. Furthermore, this general objective function can be formulated by defining a generalized value for each individual allocation unit, such as a job, or job step.

> Definition: For each program unit x_i there is an associated generalized value c_i, such that $c_i = G(a_{1i}, a_{2i}, \ldots, a_{ki})$ for $i = 1,2,\ldots,n$, where a_{ji}, $j = 1,2,\ldots,k$ are the individual attributes of program i and G is any well defined function or a composite of functions.

As is defined, the function G is perfectly general and C_i depends on program attributes which may be tangible or intangible. As an example, we may choose G to be a linear functional. Specifically, we will consider:

$$G = \delta_1 F_1(a_{1i}) + \delta_2 F_2(a_{2i}) + \ldots + \delta_k F_k(a_{ki})$$

and δ_i being either 1 or 0

for $i = 1, 2, 3, \ldots, n$. (5.2.12)

For the simplest case, suppose we consider the value to be a function of only one parameter. That is:

$$\delta_1 = 1 \quad \delta_j = 0 \quad \text{for } j = 2, \ldots, k$$

then

$$c_i = F_1(a_{1i}) \quad \text{for } i = 1, 2, \ldots, n \tag{5.2.13}$$

If $a_{1i} = \lambda_i$, i.e., the size of the program x_i and if we choose to define the function $F_1(a_{1i})$ as

$$F_1 = \frac{\alpha}{a_{1i}} \tag{5.2.14}$$

where α is a constant, then the generalized value c_i so defined is inversely proportional to the size of program x_i. When we set up our linear objective function as

$$\text{Maximize: } c_1 x_1 + c_2 x_2 + \ldots + c_n x_n$$

we are, in effect, getting a maximum degree of multi-programming as a result. This is easy to see when we realize that the value-to-size ratio in this case is inversely proportional to program size, and a descending order of value/size ratio means an ascending order of the program size. When using largest ratio first algorithm, if this is treated as continuous variable case, or using the sub-optimal algorithm in Figure 5.2 as in integer case, the programs will be loaded in a sequence with the smallest being the first. From Proposition 5.2.1 we know this will maximize the number of programs in memory.

If we conclude the possibility that each job has been assigned a priority class, it is quite natural to incorporate the priority scheme into our framework of the generalized value concept. Consider that a_{1i} being the size of program x_i and $F_1(a_{1i})$ as being defined by (5.2.14). We may consider that a_{2i} is the index to a certain priority class to which program x_i belongs. Furthermore, let us define:

$$F_2(a_{2i}) = \beta_i, \quad i = 1, 2, \ldots, n \tag{5.2.15}$$

where β_i is a constant. Then the generalized value c_i is:

$$c_i = F_1(a_{1i}) + F_2(a_{2i}) \quad i = 1, 2, \ldots, n \tag{5.2.16}$$

where

$$F_1(a_{1i}) = \frac{\alpha}{a_{1i}}$$

$$F_2(a_{2i}) = \beta_i \tag{5.2.17}$$

$$\delta_i = \delta_2 = 1, \quad \delta_j = 0 \quad j = 3, 4, 5, \ldots, k$$

With no loss of generality, we will consider α, β_i to be integers and that:

$$F_1(a_{1i}) = \left\lceil \frac{\alpha}{a_{1i}} \right\rceil, \quad i = 1, 2, \ldots, n \ . \tag{5.2.18}$$

Let us consider Table 5.7. From our previous discussion, it should be clear that $a_{11} = 20$, $a_{21} = D$, etc. If we adopt (5.2.18) as a definition for F, then we have, for $\alpha = 300$(Table 5.8). The function $F_2(a_{2i})$ will be defined by a set of integers found in Table 5.9. It can be readily verified that combining Table 5.8 and 5.9, i.e., $c_i = F_1(a_{1i}) + F_2(a_{2i})$, would give rise to each corresponding value as shown in Table 5.4.

We can see that the priority class is a way to designate a certain "urgency", based on whatever predetermined guidelines that the system employs, to each individual program. This, by itself, is artificial and the artificially chosen number, β_i, is a reflection of this perception. The choise of α is also somewhat arbitrary, so that the absolute values of both $F_1(a_{1i})$ and $F_2(a_{2i})$ are compatible, e.g. of the same order of magnitude. But this is only one of the possibilities. We can just as easily assign the β_i's to be at least one order of magnitude larger than those of $F_1(a_{1i})$'s.

In so doing, the allocation policy becomes strictly priority oriented. A more balanced approach can be implemented readily by simply adjusting the relative magnitudes between functions F_1 and F_2. In general, we may remark that the generalized value of an individual program is the composite of a set of functions whose parameters may be chosen from intrinsic program properties, such as its size, or subjective reasons, such as priority classes, or both.

Program	Size	Priority Class
x_1	20	D
x_2	25	C
x_3	18	E
x_4	27	B
x_5	30	A
x_6	15	F

Table 5.7 Example

x_i	a_{1i}	$F_1(a_{1i})$
1	20	15
2	25	12
3	18	17
4	27	12
5	30	10
6	15	20

Table 5.8 Calculated Values for $F_1(a_{1i})$ with $\alpha = 300$

x_i	a_{2i}	$F_2(a_{2i})$
1	D	25
2	C	42
3	E	14
4	B	49
5	A	60
6	F	5

Table 5.9 Assigned Real Values of Each Priority Class

5.3 Memory Utilization - Dynamic Planning

5.3.1 General Discussion

In the previous section we have examined the problem of program loading. We have, in fact, treated such action in a static manner. It is static in the sense that our goal is achieved by following an optimizing plan for that particular instance, namely, the instance of the operating system's allocating activity. We have grouped all such activities into "concentrated" points in time. But in a multiprogramming environment, not all programs terminate at the same time. Each time a program (or a particular step of a program) is done, the system either removes this program or continues onto its next job step. If a program is being removed, then memory space it once occupied would be available, thus making it possible for other programs waiting on the system job queue to be loaded, i.e., to be allocated memory space. Consequently, the activities of loading and removing jobs are interspersed throughout the system up time. Even if we may be assumed to have loaded all the programs into memory to the extent possible, the termination, and hence the removal, sequences for programs cannot be predicted, due to the fact that the system ready queue is managed in a dynamic fashion, as was discussed in Chapter 4. This dynamic management of the system ready queue constitutes what is considered to be the scheduling activity. In this context, scheduling should not be confused with allocation; one does not necessarily imply the other and vice versa. Furthermore, we have equated scheduling to execution sequences for programs. If we again consider the idealized situation in that all the program loading activities are "concentrated" and that no program will be removed individually until most (maybe all) are completed (terminated), then the two major operating system activities are, in effect, taking place cyclically. If each of such complete cycles is called a period, then we may ask what sort of planning action can we make, i.e., loading and executing, so that an objective function is optimized over several periods? Before supplying answers to this question, we must first decide upon the objectives. Of course, the memory spaces as necessary and scarce resources are central to this question. Now, it can be restated: how can we best utilize a given amount of memory space in a given processing environment? The loading (allocating) and scheduling (executing) activities thus become the means to an end. This brings up a well known model in mathematical programming applications, namely, the

Warehousing Model [38]. Essentially, the warehouse model is dealing with the question of, given a fixed capacity and the buying and selling prices of commodities over several periods of time, what action should the warehouse owner take so that his profit is maximized. In the ensuing discussions, we will examine this question in the context of operating systems.

5.3.2 Formulation for the Identical Programs Case

Let us consider a multiprogramming system where all the users (programs) are identical in size. This is neither an over simplification nor too far fetched a situation. Many so called express job queues are prime examples of this category, in which every user program is (normally) given a fixed, equal amount of memory space. The user programs are identical to one another in size only, not the amount of computations.

We will denote a list of variables as the following:

Let x_i be the total memory space occupied during period i,
y_i be the total removed memory space during period i,
M be the total memory space,
I be the initial occupied memory in period 1,
d_i be the cost per unit memory during period i,
g_i be the gain per unit memory during period i.

While some of the variables are self-explanatory, others will need further clarifications. The variable x_i is in fact the sum total of all the memory requirements for programs that are loaded in period i and y_i represents the total amount of memory space being freed up, due to the termination of programs during period i. The other two variables, d_i and g_i, are artificial quantities. We may think of the memory space as being the necessary resource for certain productive activities and that it incurs a cost when being occupied; and the system accrues profit (gain) when programs are being run to completion and subsequently removed from memory.

We will further state the following constraints:

(1) cannot occupy more memory (program loading) than what is available in any period.
(2) cannot remove (program processing) more memory spaces in period i than what was occupied in (i - 1)th period.
(3) nonnegativity, i.e., $x_i, y_i \geq 0$.

We have pointed out earlier that we view this as two major activities taking place in a cyclic manner. To "start" our problem, we must designate one of the two activities as the starting point in the model. Let us therefore assume that, initially, memory is loaded with programs, up to I units. Thus, we have arbitrarily fixed the processing activity to be the beginning activity in period 1; loading activity would follow, thus completing period 1, etc. The detailed derivation of the relations among the variables is as follows:

For i = 1, i.e., period 1, we have initially I units being occupied, and we cannot process more than this amount. Hence

$$y_1 \leq I.$$

If y_i is actually the amount of space freed, due to the completion of processing, then the total space available at this time becomes

$$M - (I - y_1)$$

and that the loading constraint states that the amount of program space being loaded cannot exceed this quantity. Therefore,

$$x_1 \leq M - I + y_1.$$

For i = 2, we have

$$y_2 \leq (I - y_1) + x_1 \quad \text{or} \quad y_1 + y_2 \leq I + x_1$$

and $$x_2 \leq (M - I) + y_1 - x_1 + y_2$$

or $$x_1 + x_2 \leq (M - I) + y_1 + y_2.$$

For i = 3, similarly, we have

$$y_1 + y_2 + y_3 \leq I + x_1 + x_2$$

and $$x_1 + x_2 + x_3 \leq (M - I) + y_1 + y_2 + y_3.$$

In general, for i = n, we have the loading constraint as

$$\sum_{i=1}^{n} (x_i - y_i) \leq (M - I) \quad , \tag{5.3.1}$$

and the processing constraint as

$$y_n \leq I + \sum_{i=1}^{n-1} (x_i - y_i) \quad . \tag{5.3.2}$$

From the definitions of d_i and g_i we see that the net gain for each period i would be

$$(g_i y_i - d_i x_i) \tag{5.3.3}$$

and it natural to state our objective over n periods to be

$$\text{Maximize:} \quad \sum_{i=1}^{n} (g_i y_i - d_i x_i) \quad . \tag{5.3.4}$$

Combining (5.3.1) and (5.3.2), we can write down the structural matrix as follows:

$$A = \begin{bmatrix}
1 & & & & & -1 & & & & \\
1 & 1 & & & & -1 & -1 & & & \\
 & \ddots & & & & & \ddots & & & \\
\vdots & & \ddots & & & \vdots & & \ddots & & \\
 & & & \ddots & & & & & \ddots & \\
1 & 1 & 1 & \cdots & 1 & -1 & -1 & -1 & \cdots & -1 \\
0 & & & & & 1 & & & & \\
-1 & 0 & & & & 1 & 1 & & & \\
-1 & -1 & 0 & & & 1 & 1 & 1 & & \\
 & & \ddots & & & & & \ddots & & \\
\vdots & & & \ddots & & \vdots & & & \ddots & \\
 & & & & \ddots & & & & & \\
-1 & -1 & \cdots & -1 & 0 & 1 & 1 & 1 & \cdots & 1
\end{bmatrix} \tag{5.3.5}$$

and let λ be the column vector of direct variables and b be the column vector of constraints, i.e.

$$\lambda = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \qquad b = \begin{bmatrix} M - I \\ M - I \\ \vdots \\ M - I \\ I \\ I \\ \vdots \\ I \end{bmatrix} \tag{5.3.6}$$

Furthermore, if we denote w to be the row vector of dual variables t_i's and u_i's corresponding to x_i's and y_i's, i.e.,

$$w = [t_1\ t_2\ \ldots\ t_n\ \ u_1\ u_2\ \ldots\ u_n] \tag{5.3.7}$$

then we can write down both the Direct formulation, and its Dual, of the Linear Programming Problem as:

(1) Direct Problem

$$\begin{aligned} &\text{Maximize:} && c\lambda \\ &\text{Subject to:} && A\lambda \leq b,\quad \lambda \geq 0 \end{aligned} \tag{5.3.8}$$

(2) Dual Problem

$$\begin{aligned} &\text{Minimize:} && (M - I)\sum_{i=1}^{n} t_i + I\sum_{i=1}^{n} u_i \\ &\text{Subject to:} && wA \geq c,\quad w \geq 0 \end{aligned} \tag{5.3.9}$$

where c is a row vector, i.e.,

$$c = [-d_1\ -d_2\ \ldots\ -d_n\ \ g_1\ g_2\ \ldots\ g_n]. \tag{5.3.10}$$

5.3.3 Special Method of Solution

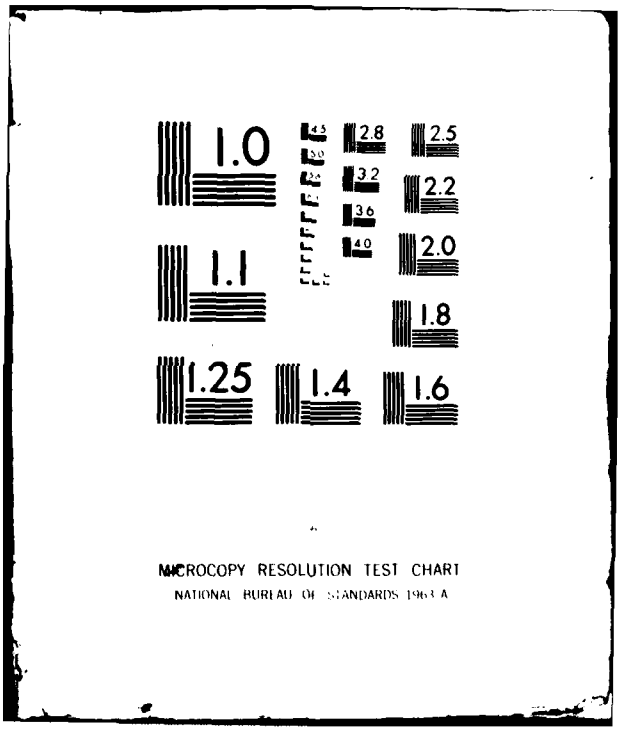
Rather than the usual Simplex Method for solving the Direct Problem, a special algorithm can be employed to attack the Dual Problem instead, due to the nature of this problem as depicted by the special form of its structural matrix shown in (5.3.5). Based on (5.3.5), (5.3.9) and (5.3.10), we can establish the following inequalities:

$$\begin{aligned}
&t_1 + t_2 + \ldots + t_n + 0 - u_2 - u_3 - \ldots - u_n \geq -d_1 \\
&\quad t_2 + \ldots + t_n + 0 + 0 - u_3 - \ldots - u_n \geq -d_2 \\
&\qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \\
&t_k + t_{k+1} + \ldots + t_n + 0 + 0 + \ldots + 0 - u_{k+1} - u_n \geq -d_k \\
&\qquad\qquad \vdots \qquad\qquad\qquad\qquad\qquad \vdots \\
&\qquad\qquad t_n \qquad\qquad\qquad\qquad\quad \geq -d_n
\end{aligned}$$

AD-A082 363 WAYNE STATE UNIV DETROIT MICH F/G 9/2
MODELLING AND RESOURCE ALLOCATION OF LINEARLY RESTRICTED OPERAT--ETC(U)
DEC 79 T FENG, C P HSIEH F30602-76-C-0282
UNCLASSIFIED RADC-TR-79-311 NL

2 . 2

END
DATE FILMED
4 80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

$$
\begin{array}{lllll}
-t_1 - t_2 - \ldots - t_n + u_1 + u_2 + & \ldots & + u_n \geq g_1 \\
-t_2 - \ldots - t_n + 0 + u_2 + & \ldots & + u_n \geq g_2 \\
\vdots & & \vdots \\
-t_k - t_{k+1} - \ldots - t_n + 0 + 0 + \ldots + 0 + u_k + \ldots & & + u_n \geq g_k \\
\vdots & & \vdots \\
-t_n & & + u_n \geq g_n
\end{array}
$$

Summarily, we have

$$\sum_{i=k}^{n} t_i - \sum_{i=k+1}^{n} u_i \geq -d_k \qquad k = 1, 2, \ldots, (n-1) \tag{5.3.11}$$

$$-\sum_{i=r}^{n} t_i + \sum_{i=r}^{n} u_i \geq g_r \qquad r = 1, 2, \ldots, n \tag{5.3.12}$$

where $t_i, u_i \geq 0$. For clarity, we will define two new variables. Let

$$\begin{aligned} T_k &= \sum_{i=k}^{n} t_i \\ U_k &= \sum_{i=k}^{n} U_i \quad . \end{aligned} \tag{5.3.13}$$

Then we can write the following relations:

$$\begin{aligned} &T_{r-1} \geq T_r \qquad r = 1, 2, \ldots, n \\ &U_{r-1} \geq U_r \qquad r = 1, 2, \ldots, n \\ &I_n = t_n \\ &U_n = u_n \\ &T_0 = T_1 \\ &U_0 = U_1 \end{aligned} \tag{5.3.14}$$

The Dual Problem can now be stated in terms of new variables:

$$
\begin{aligned}
&\text{Minimize:} && (M - I)\, T_1 + I U_1 \\
&\text{Subject to:} && \text{(a)} \quad T_k \geq U_{k+1} - d_k \\
& && \text{(b)} \quad T_n \geq -d_n \\
& && \text{(c)} \quad T_k \geq T_{k+1} \\
& && \text{(d)} \quad U_{k+1} \geq T_{k+1} + g_{k+1} \\
& && \text{(e)} \quad U_k = U_{k+1} \\
& && \text{(f)} \quad T_r,\ U_r \geq 0
\end{aligned}
\qquad (5.3.15)
$$

where $k = 1, 2, \ldots, (n-1)$; $r = 1, 2, \ldots, n$.

Clearly, from the objective function, we see that a minimum is achieved by finding the smallest possible values of T_1 and U_1. From constraints (a) through (f) in (5.3.15), that T_1 and U_1 can be found alternately, starting with T_n and U_n and then indexing in reverse order. In each step, we obtain the smallest possible values for T_k, U_k, i.e., bounded from below, according to the appropriate constraints. The process is briefly illustrated as follows:

From (b), (f) of (5.3.15),

$$T_n = \text{Max}\ \{-d_n,\ 0\};$$

From (d), (f),

$$U_n = \text{Max}\ \{(T_n + g_n),\ 0\};$$

From (a), (c), (f),

$$T_{n-1} = \text{Max}\ \{T_n,\ (U_n - d_{n-1}),\ 0\};$$

From (d), (e), (f),

$$U_{n-1} = \text{Max}\ \{U_n,\ (T_{n-1} + g_{n-1}),\ 0\};$$

$$\vdots$$

$$T_1 = \text{Max}\ \{T_2,\ (U_2 - d_1),\ 0\};$$

$$U_1 = \text{Max}\ \{U_2,\ (T_1 + g_1),\ 0\}.$$

The entire procedure for solving the minimizing problem of (5.3.15) is depicted in the form of a flow chart in Figure 5.3.

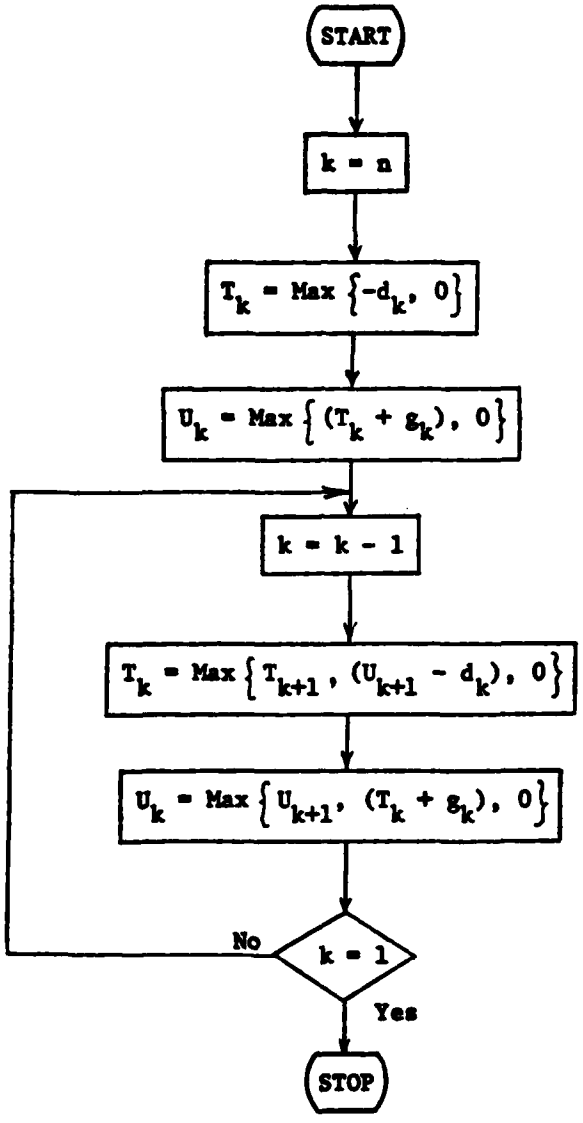


Fig. 5.3. Flow chart of the procedure for solving the special minimizing problem

5.3.4 A Special Class of Solutions

Since, at each stage, the T_i's and U_i's are determined from two or more values, a general solution in terms only of the literals of g_i's and d_i's cannot be obtained unless the specific relations between then are known beforehand.

In each step, there is always a zero as a possible lower bound, due to constraint (f) or (5.3.15). This is really redundant because there is always a T_i or U_i in the possible choices for T_{i-1} or U_{i-1} with the exception of T_n and U_n. Also be (f), we have T_i, $U_i \geq 0$. Furthermore, d_i's and g_i's are always positive. Even though we have stated earlier that these are artificial quantities, it would not be reflecting any reality to consider any negative cost and negative gain. Combining these "natural" conditions, we may obtain a special class of solutions by imposing some additional conditions,

$$g_i \geq d_i \qquad (5.3.16)$$
$$g_{i+1} \geq d_i$$

i.e., the gain per unit memory for the current and next period is at least as much as the cost per unit memory for the current period.

From constraint (a) of (5.3.15), we have

$$T_k \geq U_{k+1} - d_k ,$$

and from constraint (d), we have

$$U_{k+1} \geq T_{k+1} + g_{k+1} .$$

Combining these two inequalities, we obtain

$$T_k \geq T_{k+1} + (g_{k+1} - d_k) .$$

Since, from (5.3.16), we know that

$$(g_k - d_k) \geq 0 .$$

This implies that if (a) and (d) are true, then (c) is always true. Furthermore,

$$U_{k+1} - d_k \geq T_{k+1} + (g_{k+1} - d_k)$$

implies that

$$U_{k+1} - d_k \geq T_{k+1} . \qquad (5.3.17)$$

From Figure 5.3, we see that

$$T_k = \text{Max}\ \{T_{k+1},\ (U_{k+1} - d_k)\}.$$

Therefore, utilizing (5.3.17), we obtain

$$T_k = U_{k+1} - d_k \qquad (5.3.18)$$

for $k = 1, 2, \ldots, (n-1)$.

Similarly, we have

$$U_k \geq T_k + g_k \geq U_{k+1} - d_k + g_k$$

which implies that

$$T_k + g_k \geq U_{k+1} \; . \qquad (5.3.19)$$

Again, from Figure 5.3, we see that

$$U_k = \text{Max } \{U_{k+1}, (T_k + g_k)\},$$

and utilizing (5.3.19), we obtain

$$U_k = T_k + g_k \qquad (5.3.20)$$

for $k = 1, 2, \ldots, (n-1)$.

The values thus obtained for the T_i's and U_i's are:

$$T_n = 0$$

$$U_n = g_n$$

$$T_{n-1} = g_n - d_{n-1}$$

$$U_{n-1} = T_{n-1} + g_{n-1}$$

$$= g_n + g_{n-1} - c_{n-1}$$

$$T_{n-2} = g_n + g_{n-1} - d_{n-1} - d_{n-2}$$

$$U_{n-2} = g_n + g_{n-1} + g_{n-2} - d_{n-1} - d_{n-2}$$

$$T_{n-3} = g_n + g_{n-1} + g_{n-2} - d_{n-1} - d_{n-2} - d_{n-3}$$

$$U_{n-3} = g_n + g_{n-1} + g_{n-2} + g_{n-3} - d_{n-1} - d_{n-2} - d_{n-3}$$

$$\vdots$$

$$T_1 = g_n + g_{n-1} + \ldots + g_2 - d_{n-1} - d_{n-2} - \ldots - d_1$$

$$= \sum_{i=2}^{n} g_i - \sum_{i=1}^{n-1} d_i$$

$$U_1 = g_n + g_{n-1} + \ldots + g_2 + g_1 - d_{n-1} - d_{n-2} - \ldots - d_1$$

$$= \sum_{i=1}^{n} g_i - \sum_{i=1}^{n-1} d_i$$

The objective function has the following value:

$$(M - I)T_1 + IU_1 = (M-I)\left(\sum_{i=2}^{n} g_i - \sum_{i=1}^{n-1} d_i\right) + I\left(\sum_{i=1}^{n} g_i - \sum_{i=1}^{n-1} d_i\right)$$

$$= M\left(\sum_{i=2}^{n} g_i - \sum_{i=1}^{n-1} d_i\right) + Ig_1 \qquad (5.3.21)$$

If we are only interested in obtaining the optimum value of the objective function, then our task is done. However, we are interested in the optimal "loading and processing" strategy which would achieve such an optimum value of the objective function. In other words, we need to know the solutions in terms of the direct value, i.e., x_i's and y_i's. If the objective function for the Direct (Maximizing) Problem is denoted by π and if its Dual (Minimizing) is denoted by E, then the Duality Theorem (see Chapter 4, Section 4.4.6) states that

$$\text{Max } \pi = \text{Min } E.$$

This is to say that the two objective functions are equal at the optimum. Therefore,

$$\sum_{i=1}^{n} g_i y_i - \sum_{i=1}^{n} d_i x_i = Ig_1 + \sum_{i=2}^{n} g_i M - \sum_{i=1}^{-1} d_i M \ .$$

By equating coefficients term-by-term, we obtain:

$$\begin{aligned} &y_1 = I; \\ &y_i = M, \quad i = 2, 3, \ldots, n; \\ &x_i = M, \quad i = 1, 2, \ldots, n-1; \\ &x_n = 0 \ . \end{aligned} \qquad (5.3.22)$$

There is a rather straightforward interpretation of this result: the optimum strategy for the best utilization of a given memory space over a period of time is simply to load program space up to the maximum capacity and then to process all programs in the memory. An immediate consequence to this result

is that the preemption of any program is not desirable since it deviates from the optimum memory utilization plan. In reality, the optimum might not always be achievable due to the fact that memory cannot always be packed full.

5.3.5 Numerical Illustration

Let us consider the memory usage question of five periods, with the cost and gain in each period as given in Table 5.10. The total memory capacity is 200 units and prior to period 1, 100 units of memory space had been occupied.

Notice that the costs are identical in every period. Also, it should be clear that these cost and gain figures have no absolute meaning; only their relative magnitudes may reflect upon our system policy. We will explain this point later on.

The detailed computations, according to the procedures outlines in the last section, are as follows:

$$T_5 = \text{Max}\ \{-d_5,\ 0\}$$
$$= \text{Max}\ \{-20,\ 0\}$$
$$= 0$$

$$U_5 = \text{Max}\ \{(T_5 + g_5),\ 0\}$$
$$= \text{Max}\ \{(0+50),\ 0\}$$
$$= 50$$

$$T_4 = \text{Max}\ \{T_5,\ (U_5 - d_4),\ 0\}$$
$$= \text{Max}\ \{0,\ (50-20)\}$$
$$= 30$$

$$U_4 = \text{Max}\ \{U_5,\ (T_4 + g_4)\}$$
$$= \text{Max}\ \{50,\ (30+40)\}$$
$$= 70$$

$$T_3 = \text{Max}\ \{T_4,\ (U_4 - d_3)\}$$
$$= \text{Max}\ \{30,\ (70-20)\}$$
$$= 50$$

Period (i)	Cost (d_i)	Gain (g_i)
1	20	25
2	20	35
3	20	21
4	20	40
5	20	50

M = 200 units

I = 100 units

Table 5.10 An example with five periods

$$U_3 = \text{Max}\ \{U_4,\ (T_3 + g_3)\}$$
$$= \text{Max}\ \{70,\ (50+21)\}$$
$$= 71$$

$$T_2 = \text{Max}\ \{T_3,\ (U_3 - d_2)\}$$
$$= \text{Max}\ \{50,\ (71-20)\}$$
$$= 51$$

$$U_2 = \text{Max}\ \{U_3,\ (T_2 + g_2)\}$$
$$= \text{Max}\ \{71,\ (51+35)\}$$
$$= 86$$

$$T_1 = \text{Max}\{T_2,\ (U_2 - d_1)\}$$
$$= \text{Max}\ \{51,\ (86-20)\}$$
$$= 66$$

$$U_1 = \text{Max}\ \{U_2,\ (T_1 + g_1)\}$$
$$= \text{Max}\ \{86,\ (66+25)\}$$
$$= 91$$

The best possible net gain for the memory system over this five periods is therefore

$$(M - I)T_1 + IU_1 = (200 - 100)\cdot 66 + 100\cdot 91$$
$$= 15700.$$

This gain is a rather artificial one. What concerns us here is the strategy with which this optimum gain, whatever that may be, can be achieved. In order to obtain this optimal program, i.e., plan, we should revert back to the direct variables.

Note that the values of d_k's and g_i's satisfy (5.3.16), hence we may utilize the result stated in (5.3.21), i.e.,

$$(M - I)T_1 + IU_1 = M\left(\sum_{i=2}^{n} g_i - \sum_{i=1}^{n-1} d_i\right) + Ig_1 .$$

Since this optimum value, of the minimizing problem, is equal to the maximizing problem, due to the Duality Theorem, we have, for $n = 5$, $M = 200$, $I = 100$,

$$\sum_{i=1}^{5} (g_i y_i - d_i x_i) = 200\left(\sum_{i=2}^{5} g_i - \sum_{i=1}^{4} d_i\right) + 100g_1 .$$

By equating coefficients term-by-term, we obtain in Figure 5.4 a processing-loading pattern which is optimal in the sense as defined by (5.3.4).

If we modify slightly the values given in Table 5.10, the resultant "program" might change accordingly. Suppose g_2 has a value of 17 instead of 35, it can be shown that

$$U_2 = \text{Max } \{U_3, (T_2 + g_2)\}$$
$$= U_3$$
$$= 71$$
$$T_1 = \text{Max } \{T_2, (U_2 - d_1)\}$$
$$= 51$$
$$U_1 = \text{Max } \{U_2, (T_1 + g_1)\}$$
$$= 76$$
$$(M - I)T_1 + IU_1 = (200 - 100)\cdot(51) + (100)\cdot(76)$$
$$= 12700$$

and the optimal pattern is shown in Figure 5.5.

We see that period 2 is inactive since, according to the criterion, it is not profitable to do any processing work. This is due to the fact that $d_2 > g_2$ and therefore $U_2 = U_3$ instead of $U_2 = T_2 + g_2$. Thus, carrying on with the solution process, g_2 is missing from all the subsequent steps and there is no g_2 term in the final minimum objective functional. Hence, when we equate coefficients with the maximum objective functional, the corresponding term g_2y_2 is equal to zero. This forces y_2 to be zero, i.e., no processing activity in period. Clearly, the above analysis holds true that if for any period i, $i \neq 1$, $d_i > g_i$, then during that period there will be no processing in the final optimum plan. In the extreme case that $d_i > g_i$ for all i, then optimal strategy is simply to process everything already in the memory, i.e., the initial load, and stop. In short it is no longer advantageous to continue to operate the memory system under such circumstances. Or, viewed differently, the entire productive system (processor and memory) in this case cannot satisfactorily carry out the processing demand according to some predetermined performance criterion.

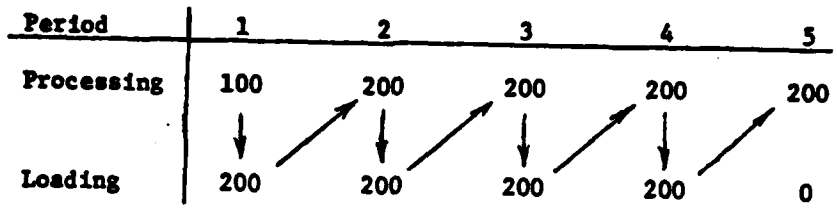


Fig. 5.4 Optimum processing-loading patterns for five periods

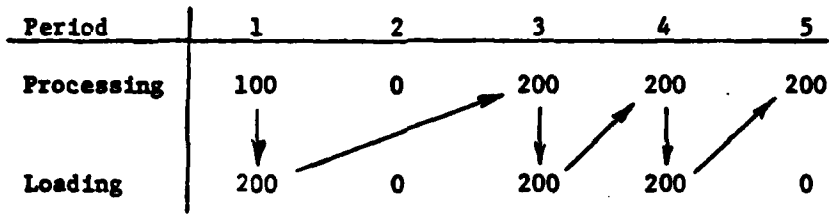


Fig. 5.5 Optimum processing-loading pattern with period 2 inactive

5.3.6 Multiprograms with Different Sizes

This represents a more typical multiprogramming environment, where different user programs have different sizes. We will, based upon the ideas and results propounded in the immediately prior sections, inquire into some possible scheduling disciplines.

Let us assume that there are m different programs loaded into memory to be processed, each with size I_i units, $\Sigma_i I_i \leq M$. If we partition the memory space exactly according to each I_i, then we may view the system as having m independent memory sections or more, each with a capacity of I_i units, with the possible exception being that portion of the memory space where no program can fit in. Furthermore, each section is full; except the fragmented portion, which is empty. Each individual section can now be viewed as similar to the problem treated in Section 5.3.2, but with only one program and such that this program takes up the entire available space.

Now we will redefine the notion of "period". Recall that in Chapter 4, Section 4.2, we briefly described the processing action of the processor in a multiprogrammed situation. A program's processing can be delayed due either to I/O activity or through timer interrupt. Therefore, the processor is being switched among all the resident programs based on either a cyclic rule or some "dynamic" discipline. If it is cyclic, then it requires no decision on the part of the system, once all the programs are in memory and the system ready queue is thus formed. However, because of the unpredictable nature in terms of timing of I/O activities, a program may not always be able to continue even though the processor has made its "round" and back again. In this case, the particular program is being "skipped" for the time being. The definition can now be stated as:

Definition 5.1: A period for program i is the time between 1) processor entering and leaving (active), or 2) leaving and returning (delayed), or 3) a skip, of program i.

Note that we consider that the processor, after entering a program and upon interrogating the condition, deciding not to stay, to be a skip. Note also that periods defined this way are possibly of unequal length within a program and among programs.

The objective function for the entire system of m programs over n periods can be stated as:

$$\text{Maximize:} \quad \sum_{j=1}^{n} \sum_{i=1}^{m} (g_{ij} - d_{ij}) I_i \qquad (5.3.23)$$

where g_{ij} is the gain for program i in period j and d_{ij} is the cost for program i in period j. Clearly, the best (optimal) strategy is that for all i, process those programs for all j such that $g_{ij} \geq d_{ij}$, and skip if otherwise. This is a direct extension of the results discussed in Section 5.3.4. If we follow this approach, then the scheduling discipline is clearly decided upon by the relative magnitudes of g_{ij}'s and d_{ij}'s. Previously, we stated that a particular program may be skipped over, possibly due to some "natural" causes such as waiting for I/O completions. By defining our objective function as (5.3.23) and following the optimal plan, it is possible to exert dynamic control over the scheduling activities by manipulating g_{ij}'s and d_{ij}'s.

5.3.7 Considerations for g_{ij}'s and d_{ij}'s and Scheduling Discipline

We have pointed out earlier that both the gain and the cost of a particular program are something rather intangible and the values chosen to quantify them are indeed artificial. However, artificiality does not imply arbitrariness. We certainly would like to consider the relevant factors in choosing their values so that, in the final analysis, the scheduling disciplines thus resulting constitute viable actions.

(1) $d_{ij} = d_{i1}$ for all j; that is, the cost per unit memory for program i does not change according to period j. Let us further denote that $d_{i1} = d_i$. We would consider this cost as a function of both the program size and memory speed, i.e.,

$$d_i = \phi(I_i, p)$$

where I_i is the size of program i and p is the speed of the memory. On first glance, it would seem redundant to include I_i as a parameter since d_i is already the cost of per unit memory. But, upon closer examination, this definition would give us the freedom to "favor" programs according to their sizes. For example, if we define d_i to be

$$d_i = \frac{\eta I_i}{p}, \quad \eta = \text{constant}$$

then the smaller program will be favored since the cost will be higher for the larger programs. Also, defined as above, d_i is inversely proportional to p, the memory speed, in units of time and the higher the speed, the higher the cost of d_i. There are, of course, many possible choices of relevant parameters and many possible functionals. We only suggest one here so as to illustrate a point.

(2) Recall that in previous sections we have discussed the "loading activity" of the system, based on the concept of generalized value c_i for program i, and we will utilize this value to start the scheduling cycle. Specifically:

i) Let $g_{i1} = c_i$

ii) If at period j, program i is being skipped over, then for period (j + 1), set

$$g_{i(j+1)} = g_{ij} + \gamma d_i \quad 0 < \gamma \leq 1.$$

This reflects the thinking that every time a program is being skipped, rather than increasing the cost of residency, we instead think of it as being potentially more valuable, i.e., higher gain, to process this program at a possible earlier time. Therefore, we increase its gain per unit memory proportional to its cost for the next period. Hence, the dynamic nature is reflected in the monotonicity of g_{ij} while d_i remains constant.

A <u>possible</u> scheduling discipline is as follows:

In period J, select the job with the highest g_{ij} among all i such that $g_{ij} > d_i$ to be processed.

We will make these remarks regarding this particular scheduling discipline:

i) It is priority influenced since $g_{i1} = c_i$ and c_i is the generalized value for program i which can be directly related to priority classes.

ii) No program will be skipped indefinitely since g_{ij} is monotonically increasing and will be processed eventually. In a way, this is dynamic readjustment on priority while the choice of c_i is static.

iii) The actual schedule depends on the function ϕ and also the constant γ. Specificity comes only with specific system.

CHAPTER VI

Program Characteristics and Memory Allocation Considerations in Page-on-Demand Systems

6.1 Introduction

In a paged environment, a task is allocated a memory size (primary) smaller than its entirety. Sooner or later, a data item will be referenced which is not found in the allocated space. This causes a page fault. A process will then be initiated to locate and bring the missing page from secondary into the primary memory. During this period, the processor is potentially non-productive; the longer and/or the more frequent this period directly reflects the deficiency of the system.

If we exclude I/O activities, then the frequency (or page fault rate) with which processor idling takes place is a function of two different categories of parameters: 1) user profile, and 2) operating system design. The duration (page wait) is a function of system configurations, i.e. bandwidth of the data channel between primary and secondary memory, and the decision of operating system design, i.e., amount of data items (page size) to be brought into primary memory. And if the secondary memory is of the rotational kind such as disc, then the time delay due to latency should also be taken into consideration.

Let us examine the two categories of parameters that influence the page fault rate.

1) User Profile - This involves the particular problems on hand (applications), the specific instruction sets available, and finally, the individual coding style. The combination of these factors produces addressing patterns that can be characterized as having certain clustering effects on various portions of the total addressing space; namely, in a given part of the program, the execution is likely to proceed in a sequential manner (maybe even jumping backward a few steps such as in a loop) for a period of time and then, when a branch instruction is encountered, moves on to a different locale which may be quite a "distance" away. Thus, if we examine the dynamic footprints (addresses) of a processor, they tend to cling together and form clusters and move from cluster to cluster and finally gravitate towards the end, i.e., termination of the program. We may look upon this phenomenon as "addressing non-uniformity". Or, equivalently speaking, the processor time is non-uniformly distributed over the addressing space.

2) System Parameters - In regard to the page fault rate, the page size is an important factor. If the size is too small with respect to the address clusters, we would expect a high page fault rate; if too large, memory utilization is low since a processor only needs those data items within a cluster most immediately.

Addressing nonlinearity is something beyond the control of system designers. The only influence we can exercise is the determination of the page size and the number of pages to be assigned to a given task. On the one hand, we would like to minimize the page fault rate, therefore high processor utilization; on the other hand, we would also like to minimize the memory waste by only loading those pages that are needed immediately and in the near future. Clearly, these two goals are conflicting with each other. In order to strike a balance between the two, page size and the number of pages loaded must be determined from the characteristics of the address clusterings. This is a brief summary of the motivation and the problem associated with the automated memory management in a hierarchical memory structure, page-on-demand computer environment. Of course, the problem will be immensely simplified if there exists a priori knowledge of the complete addressing pattern of each individual program; this is hardly a reality, however. After all, in the core of the user profile is the matter of personal coding style which is such an intangible entity. We have already touched upon the subject of dynamic processor footprints and address nonlinearity in general, Working Set in particular, in Chapter 3. In this chapter we will expound on the subject of possible analytic representation on this nonlinear property. That is, an analytic function which characterizes the mean time between page fault, in a demand paging environment. Possible memory allocation policy based on this characterization is explored.

6.2 Address Nonlinearity, Program Locality and General Lifetime Function

Belady [39] first proposed that address nonlinearity, or program locality, is the total range of storage references during a given execution interval. Denning [30] has termed, in a demand paging machine, the collection of pages in a given execution interval to be the Working Set. Due to this reference nonuniformity, it is to be expected that the page fault rate, or alternately, the mean time between page fault as a function of allocated space, assumes a nonlinear relationship. This nonlinear property

has been observed and reported by Belady [39] and others [40,41]. In order to formulate and to study analytically some of the system parameters under space sharing, Belady and Kuehner [42] proposed an analytic function, the Lifetime Function, which describes the nonlinear relationship between the page fault rate and allocated space. The following discussion will pursue this idea for the most part.

6.2.1 Lifetime Function and Some of Its Properties

Figure 6.1 depicts an "observed" relationship between the averaged processor busy-period as a function of the allocated memory space. The busy period is the time between two consecutive page faults; I/O interrupts are not represented.

S_R represents the total space required to load the entire program, hence no page faults, and $\overline{t}_R$ is the execution time for the program. There are two more interesting points on this curve, that of P and Q. For a memory allocation less than S_P, the value corresponds to P, the corresponding processor busy period, i.e., mean time between page faults, is very short, therefore resulting in high paging rates. This is the OP region. Within the PQ region, that is if the memory space falls between S_P and S_Q, the length of the processor busy period increases rather rapidly as the amount of memory space allocated increases. If the allocated space S increases beyond S_Q, the QR ragion is reached and the rate of increase in $\overline{T}$ slows down as compared to the PQ region. The behavior of the Lifetime Function in the QR region is perhaps less intuitive than that of the OP region. However, if we view this from the concept of address clusterings, it should become quite clear. As discussed earlier, most of the time during program execution, the instructions being executed tend to bunch together. The most obvious example is the DO-loop type of execution sequences. The entire execution of a program can be looked upon, in an abstract sense, as being the dynamic foot-prints of the processor which goes through a string of address clusterings. Also, there are differences in the size of the individual clusters. Let us say that the allocated space S_P is smaller than the sizes of most of the address clusterings. Then, clearly, the processor will be interrupted more frequently, due to missing items from the primary storage. On the other hand, if S_Q is greater than the sizes of most of the clusters in a program, then any space allotment beyond that point would not significantly lower the probability of page faults, and therefore a lower rate of

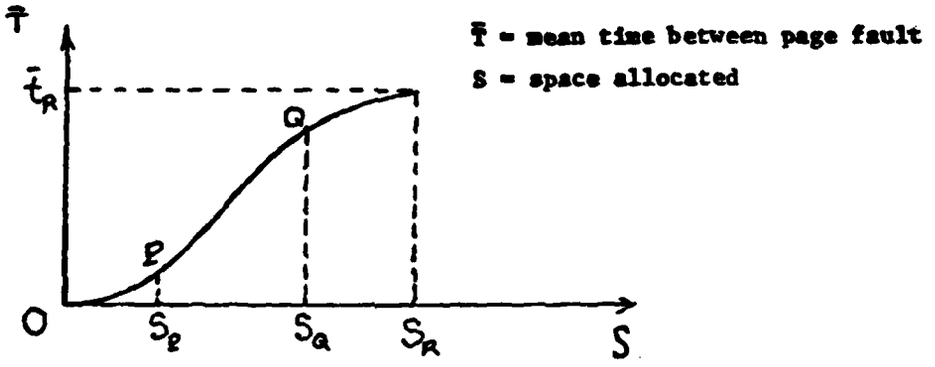


Fig. 6.1 General Observed Relationship Between Processor Busy Period And Allocated Space

increases in the average length of processor busy period, $\overline{T}$.

If we were to utilize these properties to come up with an algorithm for memory assignment, clearly we must reckon with these two special points, P and Q. Let us call them the "knees" of the Lifetime Function. Their corresponding memory sizes, S_P and S_Q, are the actual parameters we should be concerned with. Before we can say that any algorithm is optimum, we must first specify what is optimality and the norm, or performance index, by which it can be measured. The most commonly pursued system resources management objectives are that of increasing processor productive period and memory utilization. Of course, these two goals are neither mutually enhancing nor conflicting; it depends upon the system policy. Generally, the way to increase utilization of the processor is to increase the degree of multiprogramming; this will also tend to shorten the turn-around time from the users' point of view. Since not all of the program addressing space needs to reside in memory at the same time, all programs are loaded in space-squeezing fashion. Given the fixed size of primary memory, the individual program memory allotments determine the degree of multiprogramming which in turn influences the processor utilization. From this brief discussion with a simplified framework, e.g., excluding I/O considerations, we see that the important parameter is the size of the memory for each initial program loading. From the Lifetime Function as portrayed in Figure 6.1, we may discern that for a memory allocation of $S \leq S_P$ is clearly undesirable since it produces high page fault rates. For $S >> S_Q$, the increase in the average length of processor busy period may not be enough to warrant the additional amount of memory invested in. We may thus decide that for an individual program, the memory space allotted to it should at least be S_P and not much beyond S_Q. Of course, we have not as yet defined precisely where these two points are on the $\overline{T}$ vs S curve, or which point on this curve should be chosen as an allocation point.

In the foregoing discussions, we have put aside the question of page replacement; the page fault rate is considered to be a function of memory space. Actually, the page replacement algorithms also contribute to the page faults in the form of return page traffic rate. The term "return traffic" was first used by Denning in the discussion of the Working Set Model. In a space-limited environment, each new page coming in must result in another page going out. If this page of information is needed at a later

time, it must be loaded back again, constituting the "returning" traffic. If the choice of the page to be replaced is poor, then the returning traffic is "relatively" heavy, and therefore the processor busy period is relatively short. This problem is not a basic program property and is, rather, the consequence of some artificial decisions, i.e., replacement algorithms. Certainly, a good decision would minimize such return traffic. A poor one could cause the system to collapse. This problem was studied by Denning [43] and also observed by some others in actual systems [44]. For the moment, we are interested in the intrinsic program addressing property as related by processor busy period and its allocated memory space. In summing up, the general remarks one can make are: The Lifetime Function exhibits some nonlinearity; the "saturation" characteristic is due to program locality, and the processor busy period is monotonically increasing as a function of memory space.

6.2.2 Analytic Representation of the Lifetime Function

The basic parameters in the Lifetime Function are discrete in nature. In fact, when one talks about memory allocation, the unit is either in blocks or pages, due to the nature of system hardware design. In attempting an analytic form, we, in effect, have treated the memory allocation as a problem with a continuous variable. This should in no way nullify the important characteristics and rather it affords us the advantage of some mathematical tools.

Before going into more specific forms, let us first consider some possible representations of the Lifetime Function under two adverse assumptions; that the program addressing pattern is either 1) purely random, or 2) strictly sequential.

1) Purely Random

Let R = total range of program addressing space

S = allocated space for that program

where $R > S$.

For a random referencing pattern, let P be the probability that the referenced item falls within the allocated space. Then

$$P = \frac{S}{R}$$

Let x be the number of consecutive references to the allocated space, then

$$Pr[x = 1] = p(1 - p)$$

= Probability of exactly one reference before it went out of the allocated range, hence, page fault.

Similarly,

$$Pr[x = (n-1)] = p^{(n-1)}(1 - p)$$

$$Pr[x = n] = p^{n}(1 - p)$$

If $\overline{T}$ is the mean time between page faults, in units of memory references, then

$$\overline{T} = \sum_i ip^i(1 - p) \quad i = 1, 2, 3, \ldots$$

$$= p(1 - p) + 2p^2(1 - p) + 3p^3(1 - p) + 4p^4(1 - p) + \ldots$$

$$= p - p^2 + 2p^2 - 2p^3 + 3p^3 - 3p^4 + 4p^4 - 4p^5 + \ldots$$

$$= p + p^2 + p^3 + p^4 + \ldots$$

$$= p(1 + p + p^2 + p^3 + \ldots)$$

$$= \frac{p}{(1 - p)}$$

$$= \frac{\frac{S}{R}}{1 - \frac{S}{R}}$$

$$= \frac{S}{R - S}$$

We have obtained an expression for the mean time between page fault as a function of allocated space. Clearly, this is a convex function. To see this, we only have to examine its derivative

$$\frac{d\overline{T}}{dS} = \frac{(R - S) + S}{(R - S)^2}$$

$$= \frac{R}{(R - S)^2}$$

which is monotonically increasing as S increases, within the meaningful range of $S < R$. The function is plotted in Figure 6.2. For $S \to R$, we $\overline{T} \to \infty$. In other words, when the entire program is in memory, there is no page fault.

2) Strictly Sequential

If the processor fetches and executes information item by item, in sequence, then the execution time before a page fault occurs will be linearly proportional to the amount of memory space allocated. Assuming that the

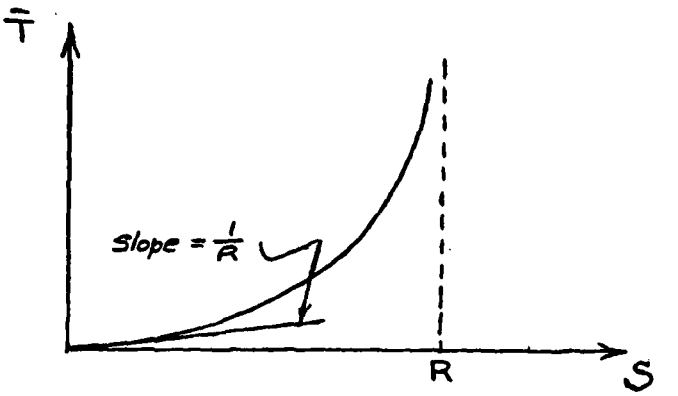
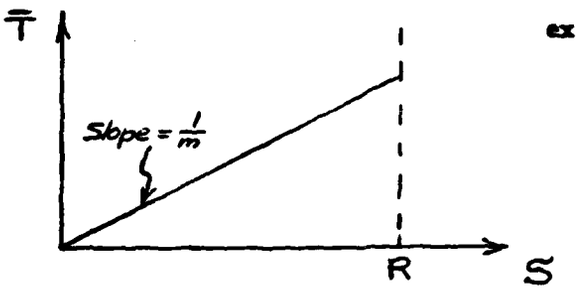


Fig. 6.2 $\bar{T}$ vs S in Purely Random Case

memory size is in units of information item per fetch, e.g., a word, or a number of bytes, then in strictly sequential case, an allocation of memory size S means that a processor can access memory S times before a page fault. The Lifetime Function would then be a simple straight line with slope of unity. In general, we can represent this function as a straight line with slope 1/m where m is the amount of information items per fetch, in memory units. For example, if we choose to represent memory size in units of bytes and in each access to memory 2 bytes will be fetched and executed, then m = 2. This is illustrated in Figure 6.3.

These two cases represent the two extremes in program addressing patterns. Of course, a real life program does not proceed in purely random fashion nor will it be in a strictly sequential manner. There are some additional properties pertaining to them which are peculiar. The mean time between page faults in the pure random case, is just what the term implies: a mathematical expectation for a given space. The replacement algorithm has no effect on its overall $\overline{T}$. In other words, page fault rate is not affected by which old page is chosen to be replaced with a new page, since the addressing is random and the probability of a page fault is only a function of the space allocated. This holds throughout the entire program execution interval. In the sequential case, $\overline{T}$ is deterministic and the replacement policy also has no effect on it; in fact, $\overline{T}$ is constant but of different value from the initial one after the first page fault. In other words, if S_1 is the initial program load size, $\overline{T}_1 = S_1/m$. If each page is of size S_2, Then after a page fault, a new page is brought in to replace an old one. The execution interval then and thereafter will always be $\overline{T}_2 = S_2/m$, irrespective of replacement policy, and $\overline{T}_1 > \overline{T}_2$ since $S_1 > S_2$.

By and large, a program would normally follow the instructions in sequence until it comes to a branch instruction. At this point there are two possibilities: branch forward or backward. A backward jump constitutes a program loop. This, to a degree, serves to reinforce the effect of address clustering, i.e., increasing the mean time between page faults for a given space allocation. A forward jump effectively reduces the portion of the program that is executed sequentially, hence, the length of the processor busy period. The picture is further complicated by the fact that it is possible to jump back after jumping forward, i.e. nested loop of sorts, and jumps out in the middle of the loop the second time around, to a point



ex: m = 2
S = 1024 bytes

Then
$\bar{T}$ = 512 references

Fig. 6.3 $\bar{T}$ vs S in Strictly Sequential Case

further down the line of the execution sequence. Branching is an inevitability with which we must reckon. For a given problem, the final program, or, more to the point, the final structure of all the "branches" is really a matter of individual style. Therefore, a desirable representation of the program Lifetime Function not only should explicitly represent its nonlinearity, i.e., the two "knees" on the curve, which is the result of address clustering, or locality, but also in some way reflects a degree of randomness, or the so-called personal styles. In summing up, perhaps we can lay down some guidelines as follows:

(1) The function should show some initial convexity and the final saturation effect due to varying sizes of the address clusterings, and

(2) It should link together $\overline{T}$, the processor busy period, and S, the storage allocated, in a simple manner, perhaps through a constant. Finally,

(3) Within the above framework, we should find as simple a functional representation as possible.

In searching for this analytic function, we should keep in mind that the chief purpose is to have a functional representation of some tangible program properties. One question confronts us immediately: should the representation be a quantitative one, or should it be qualitative one? It would be most satisfying if we can have both. But it is not to be and the reasons are simple: the paucity of data, together with the nature and the manner in which they are obtained, preclude a meaningful quantitative representation, due to the high degree of inaccuracy of the data themselves. On the other hand, there are certain observed characteristics which appear to be intrinsic to programs in general. Therefore, we opt for a general model rather than one which only suits some specific numerical values. To solidify the direction of search, we would like to have an analytic function, $\overline{T}(S)$, which satisfies the following:

(1) $\overline{T}(0) = 0$

(2) $\lim_{S \to \infty} \overline{T}(S) = \text{constant}$

(3) $\overline{T}'(S) \geq 0$, all S

(4) there exists a finite value, S_L, such that $\overline{T}''(S_L) = 0$

(5) $\bar{T}''(S) = +$ for all $S < S_L$

$\bar{T}''(S) = -$ for all $S > S_L$; S finite

Note that $\bar{T}'$ and $\bar{T}''$ are first and second derivatives with respect to S.

The function

$$\bar{T}(S) = K \frac{S^2}{S^2+D^2} \tag{6.2.1}$$

where K, D are constants, has all five properties as stated.

Let us consider the power series expansion of the function $\bar{T}(S)$.

$$\bar{T}(S) = K \frac{S^2}{S^2+D^2}$$

$$= K \frac{(\frac{S}{D})^2}{1+(\frac{S}{D})^2}$$

$$= K\,[r^2 - r^4 + r^6 = r^8 + \ldots], \quad \text{where } r = \frac{S}{D} \ .$$

The series converges to $\bar{T}(S)$ for $r < 1$, i.e., $S < D$. If $S \ll D$, then a good approximation would be

$$f(S) = K\,(\frac{S}{D})^2$$

$$= \frac{K}{D^2} S^2$$

$$= K'S^2$$

This is in agreement with the approximation chosen for the Lifetime Function by Belady and Kuehner [42]. Specifically, for a portion of the "observed" curve, namely small S, they have chosen the following:

$$f(S) = aS^b$$

where a, b are constants. For most programs, b is found to be around 2. To be more precise, $a = a'a''$ where a' reflects the property of a given instruction set and a'' is the conversion factor from storage cycles to real time, if time is indeed the unit used. If one chooses instead to consider the mean number of memory references before a page fault, called Mean Head Way by Saltzer [45], the differences will merely be a constant

conversion factor. Clearly, this approximation is valid only for S << D, where D is a constant influenced by user style, instruction set and applications, etc.; it does not account for the "saturation effect" discussed earlier, namely, the QR region of the Lifetime Function.

6.2.3 Some Limitations

The function as given by Eq. 6.2.1 does not give correct representations at the extreme values of S. According to the functional form, $\overline{T} \to K$ as $S \to \infty$. But when the storage space is large enough to contain the entire program, there is no page fault. Alternatively speaking, the mean time between a page fault, i.e., Lifetime, is infinite; in other words, the function should become singular. And we cannot interpret the value K as the total run time of the program; but we may treat K as the order of the mean time between page fault if the amount of storage allocated to a program is very near its total size.

Secondly, when $S \to 0$, $dT/dS = 0$. If we restrict ourselves to examine a very narrow range of any given program, at any given part of the program, the execution pattern is likely to be sequential. From the previous analysis, in a strictly sequential addressing pattern, the mean time between page fault is linearly proportional to the amount of storage given, with a slope inversely proportional to the processor data fetching rate which is certainly not zero anywhere.

6.3 A Procedure for Curve Fitting

In our chosen form of the Lifetime Function, constants K and D actually become parameters in characterizing individual programs and they vary from program to program. If the function is to be a useful representation, we should be able to determine the values of K and D when a set of real life data (i.e., measured from actual program) are available.

To judge whether a particular "fit" is the best fit, we must have a criterion. The most commonly employed norm is the "least square error". The procedure can be summarized as follows:

Suppose f(K, D, S) is the intended function and for each S_i there is an observed value $\overline{T}_i$. Then:

1. Form individual error term as

$$\varepsilon_i = \overline{T}_i - f(K, D, S_i)$$

2. Form the sum of error square

$$M = \sum_{i}^{N} \epsilon_i^2$$

$$= \sum_{i}^{N} [\bar{T}_i - f(K, D, S_i)]^2$$

where N is the number of data points.

3. Find K and D such that M is a minimum.

Step 3 is a generalized statement. In actual practice, there are many ways to "minimize" a function. The simplest way to locate an extreme is by setting the first derivative equal to zero.

Now consider the curve fitting problem of:

$$\bar{T} = K \frac{S^2}{S^2 + D^2}$$

From a series of $\bar{T}_i$, the observed values, corresponding to S_i, we would like to determine K and D in the least square sense.

Note that one of the parameters, namely D, is not linear. Therefore, a common strategy is by iteration as follows [46]:

i) To start the iteration, first guess the nonlinear parameter D

ii) Find best fit K, the linear parameter, in the usual manner of least square

iii) Form residual (error)

$$M = \sum_{i}^{N} (\bar{T}_i - K \frac{S_i^2}{S_i^2 + D^2})^2$$

iv) Find gradient

$$\frac{\partial M}{\partial D} = 4 \sum_{i}^{N} (T_i - K \frac{S_i^2}{S_i^2 + D^2}) \quad [\frac{K S_i^2 D}{(S_i^2 + D^2)^2}]$$

$$- \frac{\partial M}{\partial D} = 4 K \sum_{i}^{N} [\frac{K S_i^2}{S_i^2 + D^2} - \bar{T}_i] \; [\frac{S_i^2 D}{(S_i^2 + D^2)^2}$$

v) Compute new value of D by stepping in the direction of $-(\partial M/\partial D)$, assume step size h. That is, $D^{(j+1)} = D^j - h \frac{\partial M}{\partial D}$ for the j-th step iteration

vi) Find corresponding new M which is a new local minimum

vii) Repeat steps from ii) on, until stopping criterion is met

For set (i) we would like to have a reasonable guess to start the iteration in the hope that iteration time might be thus shortened. We shall first look at the derivatives of our function.

$$\bar{T} = K \frac{S^2}{S^2 + D^2}$$

$$\frac{d\bar{T}}{dS} = K \frac{(S^2 + D^2)\cdot(2S) - 2S\cdot(S^2)}{(S^2 + D^2)^2}$$

$$= K \left[\frac{2S^3 + 2D^2 S - 2S^3}{(S^2 + D^2)^2}\right]$$

$$= 2KD^2 \frac{S}{(S^2 + D^2)^2}$$

$$\frac{d^2\bar{T}}{dS^2} = 2KD^2 \left[\frac{(S^2 + D^2)^2 - 2S\cdot(S^2 + D^2)\cdot(2S)}{(S^2 + D^2)^4}\right]$$

$$= 2KD^2 \left[\frac{(S^2 + D^2) - 4S^2}{(S^2 + D^2)^3}\right]$$

$$= 2KD^2 \left[\frac{D^2 - 3S^2}{(S^2 + D^2)^3}\right]$$

$$\frac{d^2\bar{T}}{dS^2} = 0 \rightarrow S_L = \frac{D}{\sqrt{3}}$$

At this point:

$$\frac{d\bar{T}}{dS} = 2KD^2 \frac{(D/\sqrt{3})}{(D^2/3 + D^2)^2}$$

$$= \frac{18K}{16\sqrt{3}} \frac{1}{D}$$

$$= \frac{9}{8\sqrt{3}} \frac{K}{D}$$

$$\neq 0$$

In other words, the portion of the curve around this point, $S_L = \frac{D}{\sqrt{3}}$ is relatively linear. Therefore, by looking at the data points collectively, the first guess at D will be based upon the "linear" portion of the intended fit.

In step (ii), for the linear coefficient of the least square best fit, it is found as follows:

$$\bar{T} = K \frac{S^2}{S^2 + D^2}$$

For a given data pair, i.e., $S_i = \bar{T}_i$, the "residual" ε_i is:

$$\varepsilon_i = \bar{T}_i - K \frac{S_i^2}{S_i^2 + D^2}$$

The sum of square ε_i:

$$M = \sum_{i=1}^{N} \varepsilon_i^2$$

$$= \sum_{1}^{N} [\bar{T}_i - K \frac{S_i^2}{S_i^2 + D^2}]^2$$

A "best fit" would be the K such that M is minimized; this happens at:

$$\frac{dM}{dK} = -2 \sum_{1}^{N} [\bar{T}_i - K \frac{S_i^2}{S_i^2 + D^2}][\frac{S_i^2}{S_i^2 + D^2}]$$

$$= 0$$

We have:

$$\sum_{1}^{N} \bar{T}_i = K \sum_{1}^{N} \frac{S_i^2}{S_i^2 + D^2}$$

Therefore:

$$K = \frac{\sum_{1}^{N} \bar{T}_i}{\sum_{1}^{N} (\frac{S_i^2}{S_i^2 + D^2})^2} \qquad (6.3.1)$$

Furthermore:

$$\frac{d^2M}{dK^2} = 2 \sum_{1}^{N} (\frac{S_i^2}{S_i^2 + D^2})^2$$

which is always positive, hence M is a minimum when K is given by (6.3.1).

Steps (iv) - (vi) are actually known as the method of steepest descent in minimization. The detailed procedures are as follows [46]:

- Given an objective function M which is to be minimized, search in the negative gradient direction until three points are found for which the middle one has the least of the three values of M. This can be done by stepping forward in equal sized steps until a larger value than the immediately preceding one is reached. If the first step has this property, we bisect the step size and try again.

- If the three values of the objective function are found to be $M_{-h}^{(j)}$, $M_0^{(j)}$, $M_h^{(j)}$ for the jth iteration where h is the step size and:

$$M^{(j)} = M\left(D^{(j)} - h \frac{\partial M^{(j)}}{\partial D}\right)$$

then the new point where the new minimum of $M^{(j+1)}$ is:

$$D^{(j+1)} = D^{(j)} - t \left.\frac{\partial M^{(j)}}{\partial D}\right|_{D = D^{(j)}}$$

where t is found by quadratic interpolation, i.e.:

$$t = \frac{-h}{2}\left(\frac{M_h^{(j)} - M_{-h}^{(j)}}{M_h^{(j)} - 2M_0^{(j)} + M_{-h}^{(j)}}\right)$$

- The new minimum of objective function M for the (j+1)th iteration is:

$$M^{(j+1)} = M(D^{(j+1)})$$

In any iterative procedure, a common problem is that of a satisfactory stopping criterion. No general criterion exists. An often used one is the difference between the values of two successive iterations of the function in question, in this case, the objective function M. Specifically, the process shall terminate if:

$$\left|M^{(j+1)} - M^{(j)}\right| < \sigma$$

where σ is a predetermined constant. Not only the iteration time depends on the choice of value for σ but also there is no assurance that if and when it finally terminates an extremum has been found, even a local one. Again, if the function is convex, a global extremum will be assured. Another difficulty is that the iterative procedure as outlined before does not guarantee even the reduction in value of M in each iteration, even

though within each step, the minimization process, i.e., the linear square best fit and steepest descent respectively, does assure minimum in either K or M. In fact, the total square error M, either increases or decreases, depending on the initial guess of the value of D. Recalling that the procedure requires guessing at a value of D first to start the iterations. Since S_L is a point where the second derivative of the Lifetime Function vanishes, we use some "eyeball judgement" in choosing the value of S_L. According to our chosen functional representation of Lifetime Function, $S_L = D/\sqrt{3}$. Therefore, effectively, we have guessed at D.

The problem of finding, and meeting, the stopping criterion manifested itself when we tried to apply those steps for curve fitting, with a specific set of data. Real life data is very scarce and hard to come by. We have on hand a set of data, i.e., an experiment performed by Boyse [47] on a FORTRAN level G compiler executing in MTS environment at the University of Michigan. Mean time between page faults, in milliseconds, were measured for a given size of storage allocation, in number of pages, in discrete steps. The data is tabulated in Table 6.1.

With some modifications to the procedures discussed earlier, we are able to carry out the curve fitting process satisfactorily and obtain K = 839 and D = 35, with the resulting Lifetime Function as the following:

$$\bar{T} = 839 \frac{S^2}{S^2 + (35)^2} \qquad (6.3.2)$$

which is shown in Figure 6.4. The details of the modifications are given in the Appendix.

6.4 A Linear Approximation

Let us consider a power series expansion of the Lifetime Function:

$$\bar{T}(S) = K \frac{S^2}{S^2 + D^2}$$

Recall that Taylor Series Expansion for function f(s) about the point S_L is given by:

$$f(s) = \sum_j a_j (S - S_0)^j$$

where:

$$a_j = \frac{f^{(j)}(S_0)}{j!} \qquad j = 0, 1, 2, \ldots .$$

Pages in Core	Mid-point (pages)	Samples	Mean Time Between Fault (MS)
0 - 9	4.5	29	7.54
10 - 19	14.5	127	43.6
20 - 29	24.5	325	205
30 - 39	34.5	57	620
40 - 49	44.5	6	415

Table 6.1 Execution Characteristics for FORTRAN Level G Compiler in MTS Environment

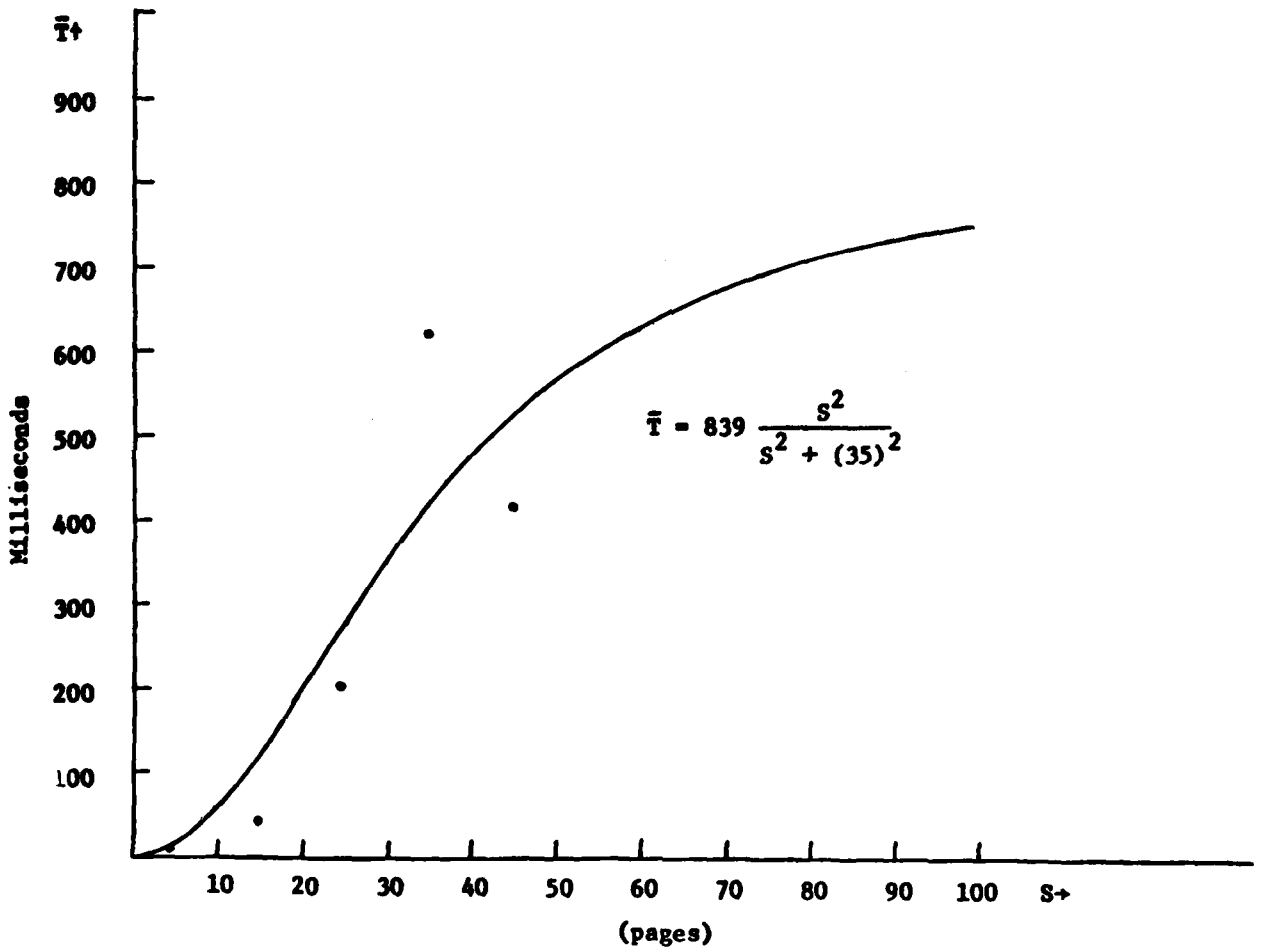


Fig. 6.4 Lifetime Function for FORTRAN Level G Compiler under MTS

If we choose to expand the function $\bar{T}(S)$ around the most linear portion, then set $S_0 = S_L = D/\sqrt{3}$. We have:

$$f^{(0)}(S_L) = K \frac{\frac{D^2}{3}}{\frac{D^2}{3} + D^2}$$

$$= \frac{K}{4}$$

$$a_0 = \frac{f^{(0)}(S_L)}{0!} = \frac{K}{4}$$

$$f^{(1)}(S_L) = K \left[\frac{(S^2 + D^2)(2S) - 2S^3}{(S^2 + D^2)^2}\right]$$

$$= K \left[\frac{\frac{2D^3}{\sqrt{3}}}{(\frac{D^2}{3} + D^2)^2}\right.$$

$$= K \frac{9}{8\sqrt{3}} \cdot \frac{1}{D}$$

$$= (\frac{3}{4})^{3/2}(\frac{K}{D})$$

$$a_1 = \frac{f^{(1)}(S_L)}{1!} = (\frac{3}{4})^{3/2}(\frac{K}{D})$$

$$f^{(2)}(S_L) = 0$$

$$a_2 = \frac{f^{(2)}(S_L)}{2!} = 0$$

$$\vdots$$

Therefore, $\bar{T}(S)$ in power series form would be:

$$\bar{T}(S) = \frac{K}{4} + (\frac{3}{4})^{3/2}(\frac{K}{D})(S - \frac{D}{\sqrt{3}}) + \ldots + R_n$$

where R_n = remainder terms, with the radius of convergance $R_c = D - \frac{D}{\sqrt{3}}$, i.e.:

$$\left|S - \frac{D}{\sqrt{3}}\right| < (D - \frac{D}{\sqrt{3}})$$

$$(\frac{2D}{\sqrt{3}} - D) < S < D$$

$$0.155D < S < D \qquad (6.3.3)$$

If we drop the remainder terms, then we have a linear approximation of the following:

$$\bar{T}(S) \simeq \frac{K}{4} + (\frac{3}{4})^{3/2}(\frac{K}{D})(S - \frac{D}{\sqrt{3}}) \qquad (6.3.4)$$

Note that:

1) If $\bar{T}(S) = 0$, then

$$K[\frac{1}{4} + (\frac{3}{4})^{3/2}(\frac{1}{D})(S - \frac{D}{\sqrt{3}})] = 0$$

$$S = \frac{D}{\sqrt{3}} - \frac{D}{4}(\frac{4}{3})^3$$

$$= 0.577D - 2 \times 0.192D$$

$$= 0.193D$$

2) For $S = 0.155D$, then

$$\bar{T}(0.155D) = K\ [\frac{1}{4} + (\frac{3}{4})^{3/2}(\frac{1}{D})(0.155D - 0.577D)]$$

$$= K\ [\frac{1}{4} - \frac{5.21 \times 0.422}{8}]$$

$$= - 0.024K$$

3) For $S = D$, then

$$\bar{T}(D) = K\ [\frac{1}{4} + \frac{5.21}{8}\ (1 - 0.577)]$$

$$= 0.524K$$

4) For $S = S_L = \frac{D}{\sqrt{3}}$, then

$$\bar{T}(\frac{D}{\sqrt{3}}) = \frac{K}{4}$$

Clearly, as it stands, (6.3.4) cannot have (6.3.3) as the valid range of S, since from (2) above, we would obtain a negative value for $\bar{T}$, which is not realistic. Therefore, we have the following linear approximation, with only the range of S slightly modified:

$$\bar{T}(S) = A + BS$$

where:

$$A = - \frac{K}{8} \qquad (6.3.5)$$

$$B = (\frac{3}{4})^{3/2}(\frac{K}{D})$$

and:

$$0.193D < S < D .$$

Some observations:

1) For a given program, if the allocated storage space is centering around A = 0.557D, where parameter D, a constant, is a program characteristic, then processor busy period between page fault, i.e., Lifetime Function, varies approximately linearly as a function of memory space.

2) For each individual program, initial allocation should be in no time less than about 20% of the value of D.

3) To invest, i.e., to allocate memory space, beyond D signifies a decrease in return; that is, the rate of increase in the processor busy period decreases as a function of space.

4) Therefore, for a given program, 0.2D to D is a "desirable" operational range for memory allocation under space-squeezing, page-on-demand environment.

5) If without additional consideration, it appears that the value of D alone suffices to render a decision on the memory allocation, for each individual program.

Let us now consider a group of user programs under paging environment. Let us further assume that we may use a linear approximation as Lifetime Function, then for each individual program i, we have, following (6.3.5):

$$\overline{T}_i = A_i + B_i S$$

Suppose we have n user programs in memory, each being allocated memory space S_i. Furthermore, let us assume that:

$$\sum_i S_i = \text{Total memory space } S_T$$

Then, the mean time between a page fault, as far as the <u>system</u> is concerned, is:

$$\overline{T}_{(system)} = \min_i \{\overline{T}_i(S_i)\}$$

If the total memory can be increased to $(S_T + \Delta S)$ and if we assume that the user programs remain to be the same set as before, and if the extra amount of memory is now distributed among these n users according

to some dividing rule, i.e.:

$$\sum_{i=1}^{n} (S_i + a_i(\Delta S)) = S_T + \Delta S$$

where:

$$\sum_{i=1}^{n} a_i = 1$$

Then the increase in $\overline{T}_i$ for each individual program due to this extra share of memory space is:

$$\Delta\overline{T}_i = a_i B_i (\Delta S)$$

Therefore, if the mean time between the system page fault before the increment in memory space is:

$$\overline{T}_{(system)} = \min_i \{\overline{T}_i(S_i)\}$$

$$= A_I + B_I S_I$$

and if, after the additional memory space ΔS, the mean time between system page fault (MTBSF) is:

$$\overline{T}^{+}_{(system)} = \min_i \{\overline{T}_i(s_i + a_i \cdot \Delta S)\}$$

$$= A_J + B_J(S_J + a_J \cdot \Delta S)$$

then the increment in MTBSF is:

$$\overline{T}_{(system} = \overline{T}^{+}_{(system)} - \overline{T}_{(system)} = A_J + B_J[S_J + a_J \cdot (\Delta S)] - A_I + B_I S_I$$

$$= (A_J - A_I) + (B_J S_J - B_I S_I) + a_J B_J (\Delta S)$$

$$= A + B(\Delta S) \qquad (6.3.6)$$

where:

$$A = (A_J - A_I) + (B_J S_J - B_I S_I)$$

$$B = a_J B_J$$

A_I, A_J, B_I, B_J, and S_I, S_J are all constants since we assume that nothing had changed except for every program there is a proportional increase in its share of the memory allottment. Clearly, $\Delta\overline{T}_{(system)}$ increases as a linear function of ΔS. This is a simplified view on the effect of increasing

the system memory capacity on the overall (i.e., system) page fault rate. This linear behavior had been observed by Saltzer on the MULTICS system at MIT [45].

6.5 Allocation Considerations

Some of the observations made in the previous section concern only the individual program. Again, when individual users competing for the limiting resources, system is the final arbitrator. In a paged environment, the system not only has to make the decisions as to which program but also the amount of that program to be loaded.

Lifetime Function as discussed here is an analytic representation of the mean time between page fault as a function of allocated memory size. If we prefer to keep the processor as busy as it can be, then, naturally, we would want to allocate as much memory as possible to a user, since the Lifetime Function is a monotonically increasing one, hence, increasing the allocated size always means an increase in the length of processor busy period. The problem is to determine what is adequate. Intuitively, we may tie this increase, or "gain" with the derivatives of the Lifetime Function. A positive second derivative always indicates a gain whenever the allocation is increased. Consequently, a negative second derivative signifies a decrease in gain if the allocation increases. From this rather informal consideration, it appears that the point at which the second derivative of the individual program's Lifetime Function vanishes would be the optimum choice for allocation. For a Lifetime Function as given by (6.2.1), this point is at:

$$S_L = \frac{D}{\sqrt{3}}$$

It may be surprising at first glance since it does not relate to the other parameter, K. A little reflection tells us that D is actually the characteristic of address clustering (a user characteristic) while K takes into account the property of a given instruction set and the conversion factor from storage cycle to real time, as was discussed in Section 6.2.2. Clearly, allocation should be based upon program properties.

The question of selecting which program to load can also be handled in like manner as that discussed in Chapter 5. Again, the problem is transformed into that of constructing the objective function using the

generalized value concept. The quantity (K/D) should be a good candidate for this consideration. Specifically, we may state our linear programming problem as:

$$\text{Maximize: } \frac{K_1}{D_1}x_1 + \frac{K_2}{D_2}x_2 + \ldots + \frac{K_n}{D_n}x_n$$

$$\text{Subject to: } \frac{1}{\sqrt{3}}\,(D_1x_1 + D_2x_2 + \ldots + D_nx_n) \leq S_T \qquad (6.3.7)$$

where S_T is the total memory units and K_i, D_i are parameters that characterize program x_i.

CHAPTER VII

Conclusions

We have studied the operating systems from a certain view point. We were influenced by the observation that the complex interactions within the computer system are quite similar to the larger economical system on a national level. If we think of the user community as a set of production requirements, each consumes a certain amount of resources and perhaps, in turn, produces commodities for consumption by the others within the society, then the analogy is clear. Cast in this light, the role of the operating system is clearly a managerial one. And therefore, all the activities within the system can be considered as cooperative efforts in achieving the production goal. The performance question thus becomes how well we can plan our production, with available, but nevertheless, limited resources. We have been quite accustomed to the notion that performance is always equated with speed, in unit time, or efficiency, in percentage of time, and the like. We may pose the question that in the total framework of production activities, whether this is the necessary and the only criterion. It is not, for we consider that this view is perhaps not general enough. For example, there is no doubt that the priority system has a reason for being in the computer system. Priority assignment sometimes can be quite arbitrary in the sense that it has little, or nothing to do with the user resource demands as against the usual approach. But we cannot gauge the loss if the system does not serve this particular user, out of no other reason than the concern for resource utilization. Once we have broadened our view on the performance question, as discussed in Chapter 3, then the usual question of optimal resource allocation can be considered as goal oriented planning (programming) and the objectives can be chosen in a very general manner.

One of the anomalies in the operating system is that one cannot quantify its design specifications. Each particular implementation entails its special characteristics. What is desirable is not some specific implementation knowledge, but rather, some general principles. In Chapter 4, we have discussed the ideas of activities and activity aggregates. This is an abstraction on the modelling of the operating system. Unlike the notion that the operating system can be modelled as a set of interacting processes, we view the system as a conglomerate of interdependent activities; inter-

dependent in the sense that they either compete for resources or their action sequences necessarily follow each other.

Memory is one of the most important resources in computer systems. By studying the memory allocation problem, coupled with the broadened view on performance, a wide range of allocation criterion is possible. We have also carefully differentiated between the allocation and scheduling problem; one is static while the other is dynamic in nature.

Finally, we consider the question of analytic representation of user program characteristics under the paging environment. From observations on the nonlinear addressing property, a possible function, the Lifetime Function, is studied. This function links together the page fault rate to that of allocated memory space. In other words, given a certain amount of memory, the page fault rate can be estimated, in principle at least, and is independent of the page replacement algorithm. It can certainly be argued that a proper (optimum) page replacement algorithm can improve on the page fault rate over this fixed Lifetime Function approach for a given memory space. This "proper" algorithm is almost certain to be a more complicated one, while the Lifetime Function approach has the virtue of simplicity. More efforts are needed in this direction in trying to understand more about user program characteristics.

In summing up, we have studied the memory allocation problem under a multiprogramming batch environment, with and without paging. The study is along general principles, rather than some specific points. Therefore, the analyses and the ideas put forth in Chapters 4, 5, and 6 should be applicable to specific systems as well. This approach is possible, even natural, after we have re-examined the performance question in Chapter 3. Constrained resource allocation problem and linear objective functions lead to Linear Programming Problems. Its mathematical underpinnings are well known. Furthermore, we have taken the general view that a system is optimal if it operates the way we want it to operate. Then the more usual notion of maximum resource utilization is but a special case. The discussion on the optimization of the objective function illuminates a certain inherent difficulty. A non-exhaustive approach in the optimizing process is therefore more realistic. We have also suggested a way, i.e., the generalized value concept, to quantify the relative importance of each individual program. Once the aforementioned "practical" algorithm for optimizing a linear

objective function is implemented in operating system modules responsible for resource allocation and program scheduling, a variety of allocation and scheduling policies can be implemented or changed, with relative ease, by changing the rules for assigning a value to each program. This approach is a departure from the conventional thinking.

APPENDIX

A. Derivation of Eq. (3.3.5)

Pure Birth Process, i.e. $\{\mu_i\} = 0$

Consider the case of constant birth rate $\lambda_j = \lambda$

Assume that the system is in E_0 at $t = 0$, then:

$$\frac{d}{dt} P_j(t) = \lambda P_{j-1}(t) - \lambda P_j(t)$$

and:

$$P_j(0) = \begin{cases} 1 & j=0 \\ 0 & j\neq 0 \end{cases} \quad P_{-1}(t) = 0; \quad j = 0, 1, 2, \ldots$$

$$j = 0, \quad \frac{d}{dt} P_0(t) = \lambda P_0(t)$$

$$P_0(t) = e^{-\lambda t} \quad \text{since } P_0(0) = 1$$

$$j = 1, \quad \frac{d}{dt} P_1(t) = \lambda P_0(t) - \lambda P_1(t)$$

$$\frac{d}{dt} P_1(t) + \lambda P_1(t) = \lambda e^{-\lambda t}$$

let: $P_1(t) = Kte^{-\lambda t}$

then: $Ke^{-\lambda t} - K\lambda te^{-\lambda t} + K\lambda te^{-\lambda t} = \lambda e^{-\lambda t}$

we have: $K = \lambda$

thus: $P_1(t) = \lambda te^{-\lambda t}$

$$j = 2, \quad \frac{d}{dt} P_2(t) = \lambda P_1(t) - \lambda P_2(t)$$

$$\frac{d}{dt} P_2(t) + \lambda P_2(t) = \lambda P_1(t)$$

$$= \lambda^2 te^{-\lambda t}$$

Note: All homogeneous solutions will be zero since $P_j(0) = 0$ for $j \neq 0$.

Again, let:

$$P_2(t) = Kt^2e^{-\lambda t}$$

$$2Kte^{-\lambda t} = K\lambda t^2e^{-\lambda t} + K\lambda t^2e^{-\lambda t} = \lambda^2 te^{-\lambda t}$$

hence: $K = \frac{\lambda^2}{2} \rightarrow P_2(t) = \frac{(\lambda t^2)}{2} e^{-\lambda t}$

Carry this further, we arrive at the solution:

$$P_j(t) = \frac{(\lambda t)^j}{j\,!}\, e^{-\lambda t} \qquad j = 0, 1, 2, \ldots$$

B. Derivation of Eq. (3.3.12)

Basic differential-difference equation

$$\frac{dP_n(t)}{dt} = \lambda P_{n-1}(t) + \mu P_{n+1}(t) - (\lambda+\mu)P_n(t)$$

for statistically equilibrium case, $P'_n(t) = 0$, all P_n's are independent of t. Then:

Subject to the constraints:

$$P_{-1} = 0 \quad \mu P_0 = 0$$

We have $\mu P_1 - \lambda P_0 = 0 \quad n = 0$

and $\mu P_{n+1} + \lambda P_{n-1} - (\lambda+\mu)P_n = 0 \quad n > 0$

The solutions are straight forward:

$$n = 0 \quad P_1 = \frac{\lambda}{\mu} P_0$$

$$n = 1 \quad \mu P_2 + \lambda P_0 - (\lambda+\mu)P_1 = 0$$

$$\mu P_2 + \lambda P_0 - (\lambda+\mu)(\frac{\lambda}{\mu})P_0 = 0$$

$$\mu P_2 + \frac{\lambda^2}{\mu} P_0 = 0$$

$$P_2 = (\frac{\lambda}{\mu})^2 P_0$$

Clearly $P_n = (\frac{\lambda}{\mu})^n P_0 = \rho^n P_0$

where $\rho = \frac{\lambda}{\mu}$ is called load or utilization factor.

C. Derivation of Eq. (3.3.16)

From Eq. (3.3.15), we have

$$\bar{L} = \frac{1-(N+1)\rho^N + N\rho^{N+1}}{(1-\rho)(1-\rho^{N+1})}$$

Suppose $\rho << 1$

$$\bar{L} = \rho(1-N\rho^N + N\rho^{N+1} - \rho^N)(1+\rho+\rho^2+\ldots)(1+\rho^{N+1}+\rho^{2N+2}+\ldots)$$

$$= \rho(1+\rho)$$

for $\rho >> 1$

$$\bar{L} = \frac{\rho-(N+1)\rho^{N+1}+N\rho^{N+2}}{(1-\rho)(1-\rho^{N+1})}$$

$$= \frac{\rho-N\rho^{N+1}+N\rho^{N+2}-\rho^{N+1}}{(1-\rho)(1-\rho^{N+1})}$$

$$= \frac{\rho-N\rho^{N+1}(1-\rho)-\rho^{N+1}}{(1-\rho)(1-\rho^{N+1})}$$

$$\simeq \frac{\rho+N\rho^{N+2}-\rho^{N+1}}{N+2}$$

$$\simeq \frac{N\rho^{N+2}-\rho^{N+1}}{N+2}$$

$$\simeq N - \frac{1}{\rho}$$

Suppose that $\rho \to 1$, we shall consider the Taylor Series expansion of $\bar{L}$.

Recall that:

$$f(x) = \sum_{k=0}^{\infty} a_k (x-x_0)^k$$

where $$a_k = \frac{f^{(k)}(x_0)}{k!} \qquad k = 1, 2, \ldots$$

In the context of our problem, it has the following form:

$$\bar{L} = \sum_{k=0}^{\infty} a_k (\rho-\rho_0)^k$$

$$a_k = \frac{\bar{L}^{(k)}(\rho_0)}{k!} \qquad \rho_0 = 1; \quad k = 0, 1, 2, \ldots$$

For $\rho \to 1$, if we take the linear approximation, then

$$\bar{L} \simeq a_0 + a_1(\rho - 1)$$

Now,

$$\bar{L} = \frac{\rho - (N+1)\rho^{N+1} + N\rho^{N+2}}{1 - \rho - \rho^{N+1} + \rho^{N+2}}$$

$$a_0 = \bar{L}(1)$$
$$= \lim_{\rho \to 1} \frac{A(\rho)}{B(\rho)}$$

where

$$A(\rho) = \rho - (N+1)\rho^{N+1} + N\rho^{N+2}$$

$$B(\rho) = 1 - \rho - \rho^{N+1} + \rho^{N+2}$$

$$A'(\rho) = 1 - (N+1)^2\rho^N + N(N+2)\rho^{N+1}$$

$$A''(\rho) = -N(N+1)^2\rho^{N-1} + N(N+1)(N+2)\rho^N$$

$$B'(\rho) = -1 - (N+1)^2\rho^N + (N+2)\rho^{N+1}$$

$$B''(\rho) = -N(N+1)\rho^{N-1} + (N+1)(N+2)\rho^N$$

Note that

$$A'(\rho) = \frac{d}{d\rho} A(\rho), \text{ etc.}$$

We have

$$a_0 = \bar{L}(1)$$

$$= \lim_{\rho \to 1} \frac{A(\rho)}{B(\rho)}$$

$$= \frac{A''}{B''}$$

$$= \frac{-N(N+1)^2 + N(N+1)(N+2)}{-N(N+1) + (N+1)(N+2)}$$

$$= \frac{-N(N+1) + N(N+2)}{-N + (N+2)}$$

$$= \frac{N}{2}$$

$$a_1 = \bar{L}'(1) \pm \lim_{\rho \to 1} \frac{d}{d\rho} L$$

$$\bar{L} = \frac{\rho - (N+1)\rho^{N+1} + N\rho^{N+2}}{1 - \rho - \rho^{N+1} + \rho^{N+2}}$$

$$\bar{L}' = \frac{A(\rho) - B(\rho)}{C(\rho)}$$

where

$$A(\rho) = (1 - \rho - \rho^{N+1} + \rho^{N+2})[1 - (N+1)^2\rho^N + N(N+2)\rho^{N+1}]$$

$$B(\rho) = [\rho - (N+1)\rho^{N+1} + N\rho^{N+2}][-1 - (N+1)\rho^N + (N+2)\rho^{N+1}]$$

$$C(\rho) = (1 - \rho - \rho^{N+1} + \rho^{N+2})^2$$

Now

$$\begin{aligned} A''' = {} & N(N+1)^2[-(N-1)(n-2)(N-3)\rho^{N-4} + N(N-1)(N-2)\rho^{N-3} \\ & + 2N(N-1)(2N-2)\rho^{2N-3} - 2N(2N+1)(2N-1)\rho^{2N-2} \\ & + N(N+1)(N+2)[N(N-1)(N-2)\rho^{N-3} - N(N+1)(N-1)\rho^{N-2} \\ & - 2N(2N+1)(2N-1)\rho^{2N-2} + 2N(2N+2)(2N+1)\rho^{2N-1}] \end{aligned}$$

$$\begin{aligned} B''' = {} & -N(N+1)[N(N-1)(N-2)\rho^{N-3} - 2N(N+1)(2N-1)(2N-2)\rho^{2N-3} \\ & + 2N^2(2N+1)(2N-1)\rho^{2N-2}] + (N+1)(N+2)([N(N-1)(N+1)\rho^{N-2} \\ & - 2N(2N-1)(N+1)(]N+1)\rho^{2N-2} + 2N^2(2N+2)(2N+1)\rho^{2N-1}] \end{aligned}$$

$$\begin{aligned} C''' = {} & 2\{-N(N+1)[(N-1)(N-2)\rho^{N-3} - N(N-1)\rho^{N-2} - 2N(2N-1)\rho^{2N-2} \\ & + 2N(2N+1)\rho^{2N-1}] + (N+1)(N+2)[N(N-1)\rho^{N-2} - N(N+1)\rho^{N-1} \\ & - 2N(2N+1)\rho^{2N-1} + (2N+1)(2N+2)\rho^{2N}] + [N(N-1)(N+1)\rho^{N-2} \\ & - N(N+1)(N+2)\rho^{N-1}] - (N+1)[-N(N-1)\rho^{N-2} \\ & - 2N(N+1)(2N-1)\rho^{2N-2} + 2N(N+2)(2N+1)\rho^{2N-1}] \\ & + (N+2)[-N(N+1)\rho^{N-1} - 2N(N+1)(2N+1)\rho^{2N-1} \\ & + (2N+1)(N+2)(2N+2)\rho^{2N}]\} \end{aligned}$$

$$\lim_{\rho\to 1} C'''(\rho) = 2\{-(N^2+N)[N^2-3N+2-N^2+N-4N^2+2N+4N^2+2N]$$
$$+ (N^2+3N+2)[N^2-N-N^2-N-4N^2-2N+4N^2+6N+2]$$
$$+ [N^3-N-N^3-3N^2-2N] - (N+1)[-N^2+N-4N^3-2N^2+2N+4N^3+10N^2+4N]$$
$$+ (N+2)[-N^2-N-4N^3-6N^2-2N+4N^3+14N^2+14N+4]\}$$
$$= 24[N+1]^2$$

$$\lim_{\rho\to 1} A'''(\rho) = (N^3+2N^2+N)[-(N^2-3N+2)(N-3)+N^3-3N^2+2N+8N^3-12N^2+4N-8N^3+2N]$$
$$+ (N^3+3N^2+2N)[N^3-3N^2+2N-N^3+N-8N^3+2N+8N^3+12N^2+4N]$$
$$= 15N^4+36N^3+27N^2+6N$$

$$\lim_{\rho\to 1} B'''(\rho) = -N(N+1)[N^3-3N^2+2N-4N(N^2-1)(2N-1)+8N^4-2N^2]$$
$$+ (N^2+3N+2)[N^3-N-(2N^2+2N)(4N^2-1)+8N^4+12N^3+4N^2]$$
$$= 13N^4+28N^3+17N^2+2n$$

Therefore

$$a_1 = \lim_{\rho\to 1} L'(\rho)$$
$$= \lim_{\rho\to 1} \frac{A(\rho) - B(\rho)}{C(\rho)}$$
$$= \lim_{\rho\to 1} \frac{A''' - B'''}{C'''}$$
$$= \frac{2N^4+8N^3+10N^2+4N}{24(N+1)^2}$$
$$= \frac{2N(N+1)^2(N+2)}{24(N+1)^2}$$
$$= \frac{1}{12} N(N+2)$$

Hence, for $\rho \to 1$

$$\bar{L} \simeq a_0 + a_1(\rho-1)$$
$$\simeq \frac{N}{2} + \frac{1}{12} N(N+2)(\rho-1)$$

D. Derivation of Eq. 3.3.17

$$(\Delta\bar{L})^2 = \sum_{i=0}^{N} i^2 P_i - \bar{L}^2$$

$$= P_1 + 2^2 P_2 + 3^2 P_3 + \ldots + N^2 P_N - \bar{L}^2$$

Note that

$$[1+4X+9X^2+\ldots+N^2X^{N-1}](1-X)^3$$

$$= (1+X)[1-(N+1)^2X^N-N^2-X^{N+1}] + 4N(N+1)X^{N+1}$$

$$(\Delta\bar{L})^2 = [\rho + 4\rho^2 + 9\rho^3 + \ldots + N^2\rho^N]P_0 - \bar{L}^2$$

$$= [1 + 4\rho + 9\rho^2 + \ldots + N^2\rho^{N-1}]\rho P_0 - \bar{L}^2$$

$$= \rho[1 + 4\rho + 9\rho^2 + \ldots + N^2\rho^{N-1}]\left[\frac{1-\rho}{1-\rho^{N+1}}\right] - \bar{L}^2$$

$$= \frac{\rho(1+\rho)[1-(N+1)^2\rho^N - N^2\rho^{N+1}] + 4N(N+1)\rho^{N+1}}{(1-\rho)^2(1-\rho^{N+1})}$$

$$= \frac{\rho(1+\rho)[1-(N+1)^2\rho^N-N^2\rho^{N+1}] + 4N(N+1)\rho^{N+1}}{(1-\rho)^2(1-\rho^{N+1})}$$

$$- \frac{\rho^2[1-(N+1)\rho^N + N\rho^{N+1}]^2}{(1-\rho)^2(1-\rho^{N+1})^2}$$

$$= \frac{\rho - (N+L)^2\rho^{N+1}(1-\rho)^2 - 2\rho^{N+2} + \rho^{2N+3}}{(1-\rho)^2(1-\rho^{N+1})^2}$$

$$\simeq \begin{cases} \rho + 2\rho^2 & (\rho << 1) \\ \frac{1}{12}N(N+2) & (\rho \to 1) \\ \left(\frac{1}{\rho}\right) + \left(\frac{2}{\rho^2}\right) & (\rho >> 1) \end{cases}$$

E. Modified Curve-Fitting Procedures

The final working procedures of Chapter 6, Section 6.3 are as follows:

1) Guess an initial value of S_L, then $D = \sqrt{3}S_L$, initial step size $h = 0.1 \times D$

2) Calculate linear coefficient K

$$K = \frac{\sum_i \bar{T}_i}{\sum_i \left(\frac{S_i^2}{S_i^2 + D^2}\right)}$$

Total Square Error:

$$M = \sum_i \left[T_i - K \frac{S_i^2}{S_i^2 + D^2}\right]^2$$

Negative gradient:

$$-\frac{\partial M}{\partial D} = 4K \sum_i \left[\frac{KS_i^2}{S_i^2 + D^2} - T_i\right]\left[\frac{S_i^2}{(S_i^2 + D^2)^2}\right]$$

3) If $\left|\frac{\partial M}{\partial D}\right| \leq 0.05$

then $h \leftarrow 100h$

$D \leftarrow D - h \frac{\partial M}{\partial D}$

repeat from step 2), otherwise continue

4) Minimize M by method of steepest descent, i.e., step off in the direction of $-\frac{\partial M}{\partial D}$ in equally spaced steps, find 3 points with the middle one the smallest, using quadratic interpolation to find new D.

5) If $|M^{(j+1)} - M^{(j)}| \leq J$; $j = 0.0001\ M^{(j)}$ then stop.

Otherwise

let $GRAD = \frac{(M^{(j+1)} - M^{(j)})}{D^{(j+1)} - D^{(j)}}$

reestablish step size

$$h = 0.1\ D^{(j)}$$

and

$$D^{(j+1)} = D^{(j)} + h \times GRAD$$

repeat from step 2) on.

This modified procedure works satisfactorily on our test problem. With different starting values of S_L, e.g., 20, 35 and 50, final values of K, D and M converge to K = 839, D = 35, M = 0.6468×10^5. In other words, we have, as a result of this curve fitting procedure, the following Lifetime Function:

$$\bar{T} = 839 \frac{S^2}{S^2 + (35)^2}$$

BIBLIOGRAPHY

[1] Rosen, S., "Electronic Computers: A Historical Survey," Computing Surveys, Vol. 1, No. 1, 7-36, March 1969.

[2] Rosin, R. F., "Supervisory and Monitor Systems," Computing Surveys, Vol. 1, No. 1, 37-54, March 1969.

[3] Denning, P. J., "Third Generation Computer Systems," Computing Surveys, Vol. 3, No. 4, 175-216, December 1971.

[4] Dijkstra, E. W., "The Structure of THE Multiprogramming System," Comm. ACM 11.5, May 1968, 341-346.

[5] Saltzer, J. H., "Traffic Control in a Multiplexed Computer System," MIT Project MAC Report MAC-TR-30, 1966.

[6] Brinch Hansen, P., "The Nucleus of a Multiprogramming System," Comm. ACM 13.4, April 1970, 238-241, 250.

[7] Dennis, J. B., "Programming Generality, Parallelism and Computer Architecture," MIT Project MAC Computation Structures Group Memo No. 32.

[8] Corbato, F. J. and Vyssotsky, V. A., "Introduction and Overview of the MULTICS System," Proc. AFIPS FJCC, Vol. 27, Pt. 1, 1966, Spartan Books, New York, 185-196.

[9] IBM, "Operating System 360 Concepts and Facilities," GC28-6535-7.

[10] Murphy, J. E., "Resource allocation with interlock detection in a multi-task system," Proc. AFIPS FJCC, Vol. 33, Pt. 2, 1169-1176. AFIPS Press, Montrale, N.J., 1968.

[11] Havender, J.W., "Avoiding deadlock in multi-tasking systems," IBM Systems Journal 2 (1968), 78-84.

[12] Wood, D. C. M., "An example in synchronization of cooperating processes: theory and practice," Operating Systems Review, SIGOPS Quarterly, Vol. 7, No. 3, July 1973, 10-18.

[13] Saaty, T. L., Elements of Queueing Theory, McGraw-Hill, 1968.

[14] Cooper, R. B., Introduction to Queueing Theory, MacMillan, 1972.

[15] Feller, W., An Introduction to Probability Theory and Its Applications, Vol. 1, 2nd Ed., John Wiley & Sons, 1962.

[16] Kleinrock, L., "Analysis of a time-shared processor," Naval Research Quarterly, 11.10, March 1964, 59-73.

[17] Rasch, P., "A queueing theory study of round-robin scheduling of time-shared computer systems," Journal of ACM, Vol. 17, No. 1, Jan. 1970, 131-145.

[18] Heacox, H. C., Jr., and Purdom, P. W., Jr., "Analysis of two time-sharing queueing models," Journal of ACM, Vol. 19, No. 1, Jan. 1972, 70-91.

[19] Coffman, E. G. and Kleinrock, L., "Feedback queueing models for time-shared systems," Journal of ACM, Vol. 1r, No. 2, Oct. 1968, 549-576.

[20] McKinney, J. M., "A survey of analytical time-sharing models," Computing Surveys, Vol. 1, No. 2, 1969, 105-116.

[21] Mitrani, I., "Nonpriority multiprogramming systems under heavy demand conditions - Customers' viewpoint," Journal of ACM, Vol. 19, No. 3, July 1972, 445-452.

[22] Kleinrock, L., "Sequential processing machines (S.P.M.) analyzed with a queueing theory model," Journal of ACM, Vol. 13, No. 2, April 1966, 179-193.

[23] Shedler, G. S., "A queueing model of a multiprogrammed computer with a two-level storage system," Comm. of ACM, Vol. 16, No. 1, Jan. 1973, 3-10.

[24] Scherr, A. L., An Analysis of Time-Shared Computer System, MIT Press, Cambridge, Mass., 1967.

[25] Anderson, H. and Sargent, R., "A statistical evaluation of the schedules of an experimental interactive system," from Statistical Computer Performance Evaluation, Walter Freiberger, Editor, Academic Press, 1972.

[26] Moore, C. G. III, "Network models for large-scale time-sharing systems," Technical Report No. 71-1, April 1971, Department of Industrial Engineering, The University of Michigan, Ann Arbor, Michigan.

[27] Hellerman, H. and Ron, Y., "A time-sharing system simulation and its validation," IBM Tech. Report, 320-2984, IBM New York Scientific Center, Data Processing Division, April 1970.

[28] Fuchs, E. and Jackson, P. E., "Estimates of random variables for certain computer communications traffic models," Comm. of ACM, 13, 12, Dec. 1970, 752-757.

[29] Boyse, J. W., "Execution characteristics of programs in a page-on-demand system," Comm. of ACM, Vol. 17, No. 4, April 1974, 192-196.

[30] Denning, P. J., "The working set model for program behavior," Comm. of ACM, Vol. 11, No. 5, May 1968, 323-333.

[31] Dantzig, G. B., "The programming of interdependent activities: Mathematical model," Activity Analysis of Production and Allocation, T. C. Koopmans, ed., New York, John Wiley and Sons, Inc., 1953.

[32] Zoutendijk, G., Twenty-Five Years of Mathematical Programming, MC-25 Information Symposium, Amsterdam, 1972.

[33] Dantzig, G. B., "Maximization of a linear function of variables subject to linear inequalities," Activity Analysis of Production and Allocation, T. C. Koopmans, ed., New York, John Wiley and Sons, Inc., 1953.

[34] Charnes, Cooper and Henderson, Introduction to Linear Programming, New York, John Wiley and Sons, Inc., 1953.

[35] Charnes, A. and Cooper, W. W., Management Models and Industrial Applications of Linear Programming, Vol. I & II, John Wiley and Sons, Inc., New York, 1961.

[36] Gale, D., Kuhn, H. W., and Tucker, A. W., "Linear programming and the theory of games," Activity Analysis of Production and Allocation, T. C. Koopmans, ed., New York, John Wiley and Sons, Inc., 1951.

[37] Little, J. D. D., Murty, D. G., Sweeney, D. W., and Karel, C., "An algorithm for the travelling salesman problem," Operations Research, 1963, 11, 972-989.

[38] Charnes, A. and Cooper, W., "Generalization of the warehousing model," Operational Research Quarterly, Vol. 6, No. 4, December 1955, 131-172.

[39] Belady, L. A., "A study of replacement algorithms for a virtual storage computer," IBM Systems Journal 5, 2 (1966), 78-101.

[40] O'Neil, R. W., "Experience using a time-sharing multiprogramming system with dynamic address relocation hardware," Proc. AFIPS 1967 SJCC, Vol. 30, 611-621.

[41] Varian, L. C. and Coffman, D., "An empirical study of the behavior of programs in a paging environment," ACM Symposium on Operating System Principles, Gatlinburg, Tenn., Oct. 1967.

[42] Belady, L. A. and Kuehner, C. J., "Dynamic space-sharing in computer systems," Comm. of ACM, 12, 5 (May 1969), 282-288.

[43] Denning, P. J., "Thrashing: Its cause and prevention," Proc. FJCC, Vol. 33, Part One, 1968, 915-922.

[44] Rodriguez-Rosell, J., and Dupuy, J.-P., "The design implementation, and evaluation of a working set dispatcher," Comm. of ACM, April 1973, Vol. 16, No. 4, 247-253.

[45] Saltzer, J. H., "A simple linear model of demand paging performance," Comm. of ACM 17, 4 (April 1974), 181-186.

[46] Hamming, R., Introduction to Applied Numerical Analysis, McGraw-Hill, 1971.

[47] Coffman, E. G., Jr., Elphick, M., and Shoshani, A., "System Deadlocks," Computing Surveys 3, 1 (June 1971), 67-78.

[48] Brinch Hansen P., Operating System Principles, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

[49] Edmonds, J., "Paths, trees, and flowers," Can. Journal of Math., 17, 1965, 449-467.